NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SCALABLE MULTICAST TREE CONSTRUCTION FOR WIDE AREA NETWORKS

by

James Eric Klinker

September 1994

Thesis Advisor:	Shridhar B. Shukla
Second Reader:	Gilbert M. Lundy

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Scalable Multicast Tree Construction For Wide Area Networks (U)	5. FUNDING NUMBERS
6. AUTHOR(S) Klinker, James Eric	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In this thesis, we address the problem of multicast tree construction with guaranteed quality of service (QoS) in networks with asymmetric link costs. We describe a protocol to locate distribution centers for an interaction based on network load and participant location. We then describe the protocol for constructing a shared tree around the selected center. We compare the quality of the resultant trees on large hypothetical networks with that of source based trees. Additional comparisons are made with other multicast techniques such as Protocol Independent Multicasting (PIM) and Core Based Trees (CBT). Our results show that the shared trees built using our approach represent a significant improvement over other techniques when the network topology contains a large degree of asymmetry in link cost. This makes our approach the most general of all other techniques proposed to date.

14. SUBJECT TERMS Multicast trees, scalable, quality of service, wide area networks	15. NUMBER OF PAGES 114
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

SCALABLE MULTICAST TREE CONSTRUCTION FOR WIDE AREA NETWORKS

by

James Eric Klinker
B.S. Electrical Engineering, University of Illinois at Urbana-Champaign, 1990

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

September, 1994

Author: [illegible]
J. Eric Klinker

Approved By: [illegible]
Shridhar B. Shukla, Thesis Advisor

[illegible]
Gilbert M. Lundy, Second Reader

[illegible]
for Michael A. Morgan, Chairman,
Department of Electrical and Computer Engineering

ABSTRACT

In this thesis, we address the problem of multicast tree construction with guaranteed quality of service (QoS) in networks with asymmetric link costs. We describe a protocol to locate distribution centers for an interaction based on network load and participant location. We then describe the protocol for constructing a shared tree around the selected center. We compare the quality of the resultant trees on large hypothetical networks with that of source based trees. Additional comparisons are made with other multicast techniques such as Protocol Independent Multicasting (PIM) and Core Based Trees (CBT). Our results show that the shared trees built using our approach reserve fewer resources than the source based trees even for a significant number of simultaneous senders. The shared trees built using our approach represent a significant improvement over other techniques when the network topology contains a large degree of asymmetry in link costs. This makes our approach the most general of all other techniques proposed to date.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank Dr. Shridhar B. Shukla for his guidance and his relentless effort in moving this work to maturity.

I. INTRODUCTION

A. BACKGROUND

The integrated packet switched networks of the future are expected to provide users with a variety of multiparty interaction capabilities. These services will benefit from a network level multicast with guaranteed Quality of Service (QoS). Guaranteed QoS, in terms of delay and jitter bounds, can be provided through the reservation of resources such as buffer and packet processing capacity at network nodes [10]. Network level multicast pioneered in [7, 8] refers to the reduction in the amount of traffic for point to multipoint (group) communication when compared with unicasts. One way to provide network level multicast requires the network to form a multicast routing tree based on the members of the group and their location [8].

There are two basic approaches to multicast tree construction. The first is a shared or center-specific tree (CST) [1] and the other is a source based or sender-specific tree (SST) [5, 7]. A center-specific approach utilizes a single tree rooted at some center that is shared by all senders. In the sender-specific approach each sender builds a separate tree rooted at itself. The center-specific tree reserves fewer resources for an interaction that contains multiple, yet non-concurrent, senders that require reservations. The disadvantages of a center-specific tree are that over large groups, certain links may become bottlenecks and in the case of a large number of concurrent senders, traffic concentration may occur [12]. Current techniques suggest that the center (core) be selected administratively. Ideally, it should be selected algorithmically based on the participants' locations and network load distribution.

The sender-specific tree approach is scalable and efficient in interactions with a large number of concurrent senders. The volume carried along each tree is the same regardless of the number of senders. The sender-specific tree makes excessive reservations when the number of concurrent senders is small compared to the total number of senders requiring guaranteed QoS. The reservations will occur along every tree although every tree does not carry traffic at the same time. Thus, depending upon the type of reservation-based interaction to be set up, either a center-specific tree or a sender-specific tree will be most economical. However, neither approach implements tree construction according to the availability of the resources required to guarantee QoS. Since resources are consumed along the routes taken by the multicast traffic, an approach that provides guaranteed QoS should couple tree construction with resource availability. The current draft-standard for resource reservation being considered by the Internet Engineering Task Force (IETF) is the Resource ReSerVation Protocol (RSVP) [14]. RSVP proposes receiver-initiated reservation of resources which are completely independent of the way the network routes packets and creates multicast trees.

B. MOTIVATION AND OBJECTIVES

The motivation for this work is as follows. Multicasting with guaranteed QoS should permit a flexible choice of center-specific trees and sender-specific trees based on the number and the nature of participants. The choice of tree type should not be left entirely up to the receiver but should be based, to the extent possible, on *a priori* information about the participants. The tree centers should be selected algorithmically based on this information. A resource availability check should be made at tree construction time to spread the load uniformly and to prevent cases of unobtainable reservations after the tree is set up. Finally, the burden of resource reservation should be shared by the senders and the receivers, particularly when the senders are numerous and relatively long-lived. Instead of making the receivers obtain reservations to every sender, such senders should obtain reservations up to some distribution center and the receivers should obtain reservations from the center.

While it is reasonable to expect that most of the links in the Internet are symmetric in their capacities, it may be unreasonable to expect the traffic load on any given link to be symmetric (consider the case of ftp, most of the bandwidth is consumed in only one direction). It is shown in [20, 21] that loads on the NSFNET backbone are not symmetric. A single-site study [22] confirmed the existence of certain "busy source" or "favorite site" effects, where a small number of hosts dominate the network traffic. These effects should contribute to network asymmetry. This is particularly true if the routing techniques used in the network do not support asymmetric topologies, since any degree of asymmetry is likely to be heightened by the routing technique. These results are supported by simulations where multiple interactions using a reverse path routing mechanism were built on a uniform topology and the resultant topology was not symmetric [15].

The specific objectives of this thesis are:

1. Define an approach for multicast data distribution that permits flexible construction of center-specific and sender-specific trees using *a priori* information about participants.

2. Determine an algorithmic technique to locate a distribution center for a center-specific tree.

3. Describe an approach for center-specific tree construction in the presence network asymmetry. The approach should be valid for symmetric topologies.

4. Compare the quality of the resultant center-specific trees with sender-specific trees.

C. ORGANIZATION

The thesis is organized as follows. Chapter II describes our approach to meet the design goals listed above. Chapter III describes the protocols for center selection and tree construction required by the approach. Chapter IV describes the performance evaluation of the approach. Chapter V details a comparison with some existing techniques. Chapter VI concludes with general results and directions for future research.

II. GENERAL APPROACH

A. USE OF *A PRIORI* INFORMATION ABOUT THE PARTICIPANTS

The requirements of a multiparty interaction are determined by the following three characteristics: reservation requirements of individual participants, the number of concurrent senders, and the location of the participants. Thus, every participant should know an estimate of its resource requirements (bandwidth), its address (location), and the nature of the role it will play in the interaction (specifically its sending/receiving requirements). We anticipate a network entity called a *scheduler*, similar to the session directory (sd) [16] tool. The scheduler can determine the requirements of the interaction if it is given the above information about potential participants. The scheduler is responsible for grouping participants into groups referred to as *critical sets of participants* (CSP). From the sending requirements of a participant, the scheduler can determine if that participant should form a sender-specific tree or join a shared center-specific tree. If the participant is expected to source traffic throughout the interaction then a sender-specific tree is most efficient. On the other hand, if the participant is a member of a group that is expected to have a single sender at any give time, then those participants should share a tree. Participants in distant and separate routing domains should not share a tree. Thus, based on *a priori* information, the combination of sender-specific trees and center-specific trees for an interaction can be controlled.

B. ALGORITHMIC LOCATION OF A SET OF DISTRIBUTION CENTERS

In this approach, there is one distribution center located for each CSP. If a CSP consists of a single participant, the designated router for that participant becomes the distribution center.

A center location mechanism should be fast, scalable, and the resultant tree should minimize the consumption of network resources. The proposed mechanism implements a tournament among selected network routers, the winner of which is determined to be the selected center for the center-specific tree. The mechanism relies on the scheduler to assign a *CSP id* to each member of the CSP and a *participant id* to each participant in the interaction. The designated routers for participants with the same *CSP id* enter into a pairwise selection process that results in the selection of a distribution center. The scheduler is responsible for the initial pairing of routers in this process. The initial pairing of routers is based on inter-participant distance since this information is available to the scheduler *a priori*. However, the locations in the subsequent phases are not known *a priori*, and, therefore, these pairings happen without any distance information in this mechanism.

The initial pairing proceeds as follows. Senders are paired with receivers until no more senders or receivers remain. If there are more senders than receivers, any remaining senders are paired with each other (or byes which may be required to make the number of teams in the tournament a power of 2). Otherwise, the remaining receivers are paired together (or with byes). An example tournament can be found in Figure 5 when the center selection protocol is described in Chapter III. Participants that are farthest apart from each other are paired together in order to rapidly move the center towards a cluster of participants. This minimizes the impact of distant participants on the final center selected. Using the same argument we want to minimize the number of successive byes that may occur. If a participant receives a bye in more than a single successive phase, that participant will have a greater impact on the final center selection.

For each pair, a router is selected (called the winner) that represents the middle of the shortest path between the pair. The process is repeated for pairs of winners until a single router remains. The router selected as the winner for the CSP informs the scheduler of the *CSP id* and *group id* for which it won. The scheduler maintains a map of *group ids,* *CSP ids*, and their associated centers. When all winners have reported, the scheduler sends the registered participants a complete list of center locations for all CSPs in that group. The protocol for the automatic location of distribution centers is given in Section A of Chapter III.

C. TREE CONSTRUCTION

Once the distribution centers are located the participants must become aware of the centers and join the tree. For a participant with a particular *CSP id*, the center with the same *CSP id* in the list supplied by the scheduler is selected as the home center. Each receiver joins all distribution centers to receive traffic, each sender need only join the home center to distribute traffic.

In current techniques, a new group member not connected to a router that is group-aware relies on some protocol like IGMP [13] to reach a router that is group-aware. In the future, we expect that the new member could rely on some hierarchical global group membership service that associates a group name with the address of the nearest distribution center.

The join process is similar to the recently proposed Core Based Trees (CBT) approach [1] with some slight differences due to the asymmetry. A sender joins a distribution center by propagating a join-request along the shortest path to the distribution center. When the join-request reaches the center, a join-ACK is sent back along the same path. A receiver joins in a similar manner by propagating a join-request to the center along the shortest path. The join-ACK is then sent back to the receiver along the shortest path (likely to be different than the path taken by the join-request). Note the above approach allows separate paths for send and receive traffic which could result in routing loops if the mechanism does not guard against it. This is illustrated in Figure 1. The solid arrows represent the paths for send traffic, the dashed arrows represent the paths for receive traffic. Note that at router 1, traffic from sender S1 is forwarded out interfaces a, b, and c but traffic from any other source should only be forwarded out interfaces a and b. To forward S2's data out c would generate a routing loop.

To accommodate this situation, the on-tree routers must maintain a source list. This source list contains the set of interfaces, per sender, that traffic, originating from that sender, should take. This explicit source list increases the state maintained per network node when the number of senders per CSP is large. It should also be noted that the resulting directed graph (as in Figure 1) in no way adheres to the graph theory definition of a "tree." However, throughout the thesis this graph will still be referred to as a multicast "tree." A detailed description of the tree construction protocol is given in Section B of Chapter III.

Dynamic membership is handled in a manner similar to CBT. A router is either on-tree or off-tree with respect to a given center-specific tree. A router that is on-tree for any center-specific tree is considered *group aware*. This router maintains a listing of all centers for a particular *group id*.

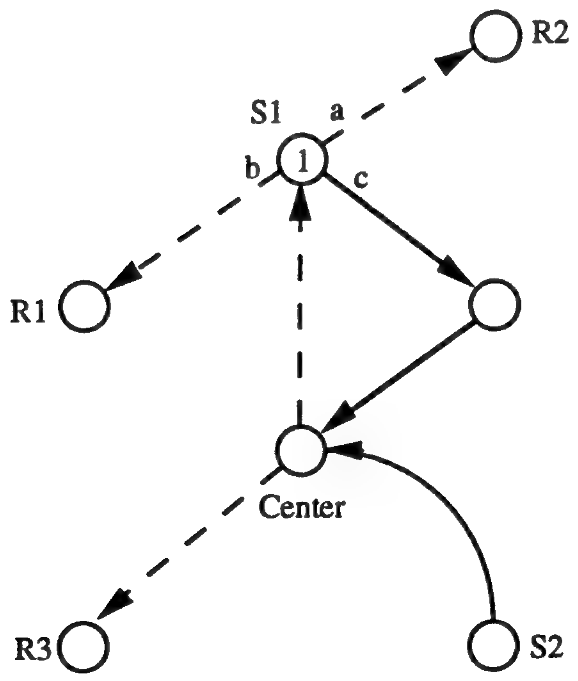


Figure 1. Tree Construction

Currently, no provision is made to relocate distribution centers during an interaction. It is assumed that the number of unanticipated senders in a multiparty interaction does not change excessively throughout the interaction. If this is not the case, a method of periodically relocating the centers is probably sufficient to handle unanticipated participants.

D. RESOURCE RESERVATION

In the ideal case, resource reservation should occur before the participants send or receive traffic on the tree. The center for the tree is located based on available resources if the pairwise selection process determines the shortest unicast path using the *unreserved bandwidth* type-of-service (ToS) option such as the one permitted by Open Shortest Path First (OSPF) [6]. Thus, the winner in each phase is selected along the path with the most unused bandwidth available. Use of ToS-based routing in this stage will make the center selection mechanism sensitive to the current network load. Similarly, if the join-requests and join-ACKs are unicast to the center using the same routing option, the branches that are grafted to the tree will contain links with the most bandwidth available. In this manner, the tree is built with resource availability as a primary consideration.

Reservations can occur when the control messages that graft a participant to the tree are sent. Senders are added to the tree via the join-request messages. Reservations can occur when the state for that sender is modified at any router. For receivers, reservations to receive traffic from all current senders can be made as the join-ACK propagates back to the receiver. Thus, the burden of reservation is shared by both senders and receivers. All reservations for a participant are made before the participant sends or receives traffic.

If sufficient resources are unavailable at the time the connection to the tree is to be made, an unreserved connection should take place. This may result in degraded service but minimizes establishment latency over an approach that waits for sufficient resources before the connection is made.

III. PROTOCOLS

Several protocols are required to support the approach. A *participant registration protocol* is required that allows participants to register their attributes with the scheduler before an interaction. A *center selection protocol* is required to implement the pairwise center selection process. A *tree construction protocol* is required to add the senders and receivers to a pre-determined center. A *reservation protocol* is required to obtain reservations on the links of the tree.

A qualitative description of the services provided by each has been given in the previous chapter. A detailed description of the center selection protocol is presented in section A. A detailed description of the tree construction protocol is presented in section B. The participant registration and reservation protocols will be described in detail in future work.

A. CENTER SELECTION PROTOCOL

This section expands on the general description of the center selection mechanism presented in Section B of Chapter II. The algorithms for determining the selection hierarchy and the bye positions in the first phase, described in Section B of Chapter II, are shown in Figures 2 and 3 respectively [23]. For each pair of participants a winner is determined by the following actions. If the pair consists of a sender and a receiver, the sender is designated as the leader of the pair and a probe is sent via a unreserved bandwidth *type-of-service* unicast along the shortest path to the receiver. This probe is echoed back to the leader along the same path and the route is recorded. From this information the leader selects the middle router along this path as the winner for the pair. Some modifications are necessary if two senders or two receivers are paired together. The participant with the lowest CSP id is designated as the leader. This participant sends a probe along the shortest path to the partner recording the route along the way. The partner then encapsulates this information in a new probe which is sent back to the leader along the shortest path (likely a different path altogether).

```
SelectionHierarchy for CSP id = $cspid$ at scheduler
    $numrps$ = number of registered participants in $cspid$;
    number of phases $n = [\log_2 numrps]$;
    /* $slots[i][j]$ is the $j$th winner in phase $i$;*/
    initialize $slots[0][j]\ \forall j \in [1, 2^n]$ with $bye$
        using ByeDetermination;
    initialize pairs of empty $slots[0][j]$ with sender-
        receiver pairs in decreasing order of distance;
    initialize remaining $slots[0][j]$ with remaining
        participants;
    for $i = 1$ to $n$
        $numslots = 2^{n-i}$ /* number of slots in phase $i$ */;
        for $j = 1$ to $numslots$
            $slots[i][j]$ = winner of $slot[i-1][2j-1]$ and $slot[i-1][2j]$;
end SelectionHierarchy
```

Figure 2. Algorithm for Determining Selection Hierarchy

The probe records the route taken. The leader then calculates the cost of the two paths and chooses the middle router along the lower cost path as the winner. For the pairing of the

```
ByeDetermination by scheduler
    number of phases $n = [\log_2 numrps]$;
    $byecount = 2^n - numrps$;
    $count = 0$;
    while $byecount > 0$
        $m = 2^{[\log 2\ (byecount)]}$;
        for $i = 1$ to $m$;
            set $slots[0][2^{n - count} - m + i]$ as a bye position;
        $byecount = byecount - m$;
        $count = count + 1$;
end ByeDetermination.
```

Figure 3. Determining the Bye Positions in the First Phase

subsequent winners, two probes are always used to determine the lower cost path along which to select a winner. This process can best be illustrated by example.

Consider the simple topology in Figure 4a with the two senders and four receivers. The links are bi-directional with asymmetric costs. Figure 5 shows the initial selection hierarchy determined by the scheduler. Each participant knows this hierarchy and from it can determine its partner and the pair of participants whose winner its winner will be paired with. The distance from S1 to R2 (10) is the greatest so this pairing is selected over all others. The next sender-receiver pairing corresponding to the next greatest distance (5) is S2 and R3. This leaves R1 to be paired with R4. The probes of the first phase are shown in Figure 4b. Each pairing is shown with a different arrow type to distinguish it from

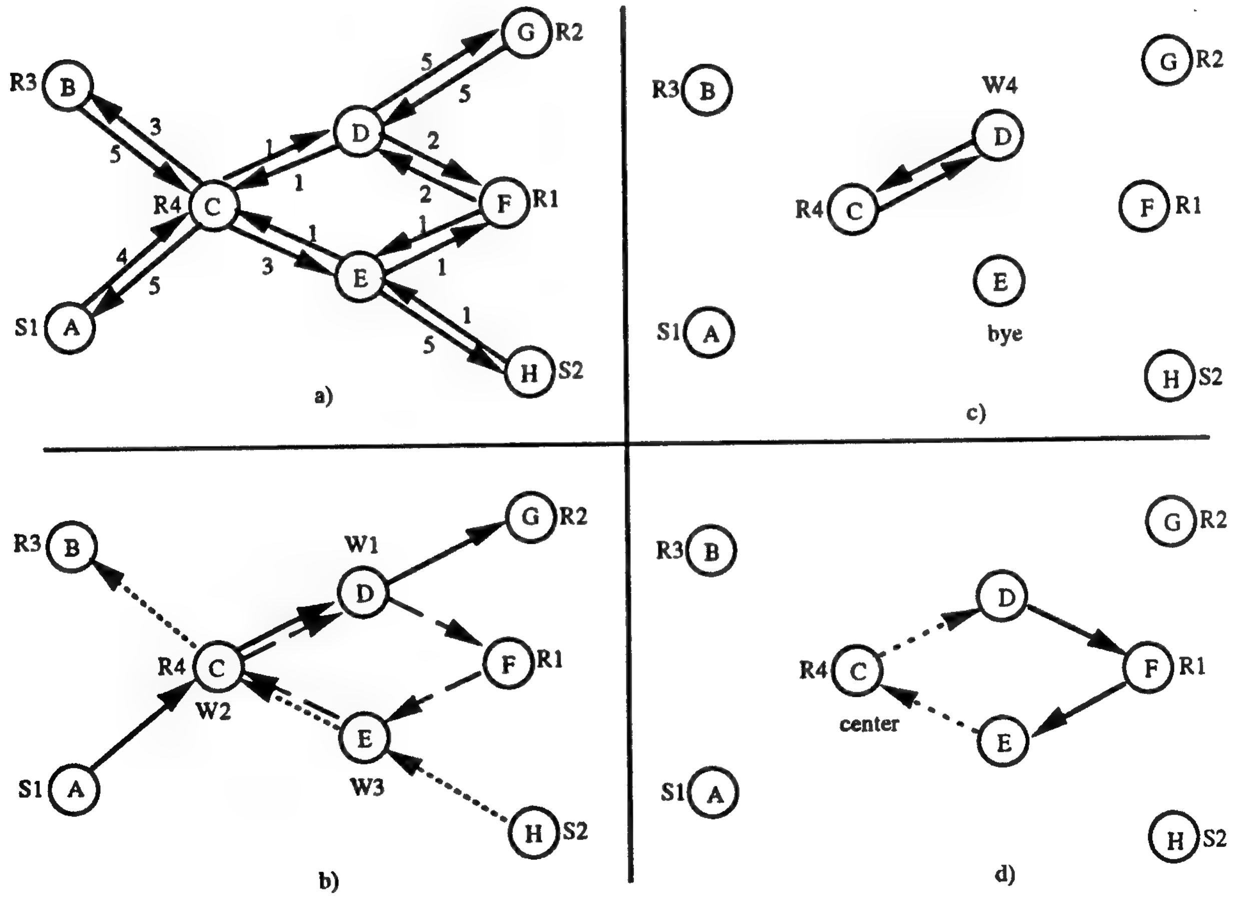


Figure 4. Center Selection Protocol Phases

the other pairs. S1 (router A) sends a probe along the shortest path to R2 (router G). Along this path router D is selected as the winner and designated with the label W1. Likewise, for the pairing of S2 and R3, router C is determined to be the midpoint of the path and is selected as the winner (W2). The pairing of R1 with R4 requires an additional probe. R1 is designated the leader and sends a probe to R4 via router E. R4 encapsulates this path in a probe that is sent back to R1 along the shortest path. This path takes the probe through router D. The cost of the first path (2) is compared with the cost of the second path (3) and router E is selected as the winner (W3). The second phase is shown in Figure 4c. The winner between routers C and D is determined to be D while router E

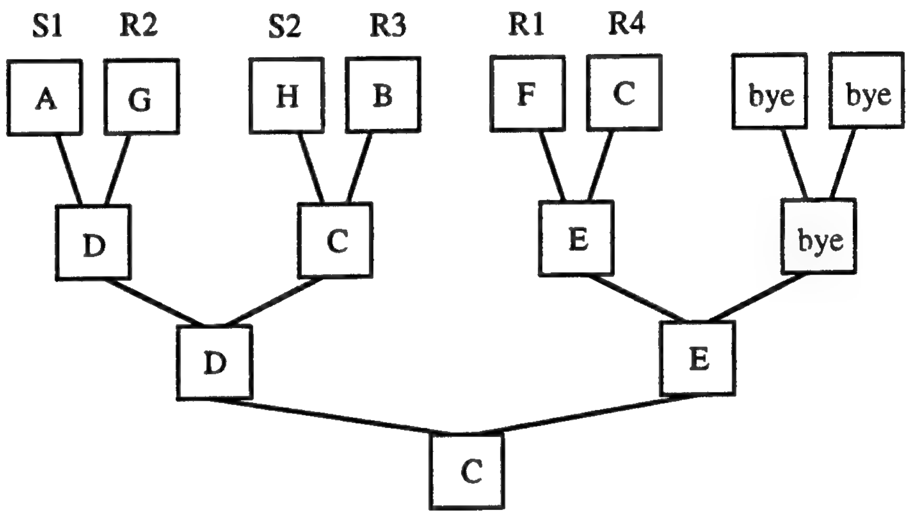


Figure 5. Selection Hierarchy

receives a bye in this phase. The final phase is shown in Figure 4d. In this phase the lower cost path is from router E to D (at a cost of 2) and router C is selected as the overall winner.

The method for determining the location of a partner is as follows. The leader of a pair informs all other leaders of the location of its winner. Since the function of all leaders is the same, every leader has a complete list of winners when the phase is over. The leader then informs its winner of the locations of all other winners along with the phase number, the CSP id, the group id, and the leader's id. So after each phase every winner knows every other winner and every winner can determine its partner in the next phase. Since the number of phases required is deterministic ($\log_2$ *numrps*) the winner of the final phase can determine that it is the overall winner and communicate its location to the scheduler along with the *CSP id* it represents.

B. TREE CONSTRUCTION PROTOCOL

```
1. Propagate the join-request to the center.

2. At each off-tree router along the path:
    Create a sender list (SL) with the new sender that contains the last hop
    as the incoming interface and the next hop as the single OGI.

3. At each on-tree router along the path:
    3.1 Add the sender to the current SL with the incoming interface = last
    hop and {OGIs} = to all OGIs for all other senders in the SL.

    3.2 Send a Build-State message out all OGIs except the OGI that
    represents the next hop to the center.
        3.2.1 If the Build-State encounters a router that does not contain the
        sender in the SL, add the sender to the SL with the incoming
        interface = the last hop, and the {OGIs} = to all OGIs for all
        other senders in the SL.
            3.2.1.1 Send a Build-State message out each of these interfaces.
        3.2.2 If the Build-State encounters a router with an entry for the
        sender in the SL:
                Prune the OGI that led to that router from the {OGIs} for the
                sender at the previous router. If the {OGIs} = { }, remove
                the sender from the SL and and prune the OGI that led to that
                router from the previous router.

                Continue pruning until an entry for the sender is encountered
                that still has at least one other OGI left. The prune message
                knows the location of the previous router from the incoming
                interface entry stored with the sender.
```

Figure 6. Protocol to Add New Senders

The method of tree construction presented in Section C of Chapter II requires a protocol to build the state in each router associated with new senders or receivers. This section details such a protocol.

The information, per sender, maintained at each router is [Sender ID, incoming interface, {set of outgoing interfaces (OGI)}]. A group and CSP id should also be attached per sender list. The protocol for adding new senders is presented in Figure 6. This

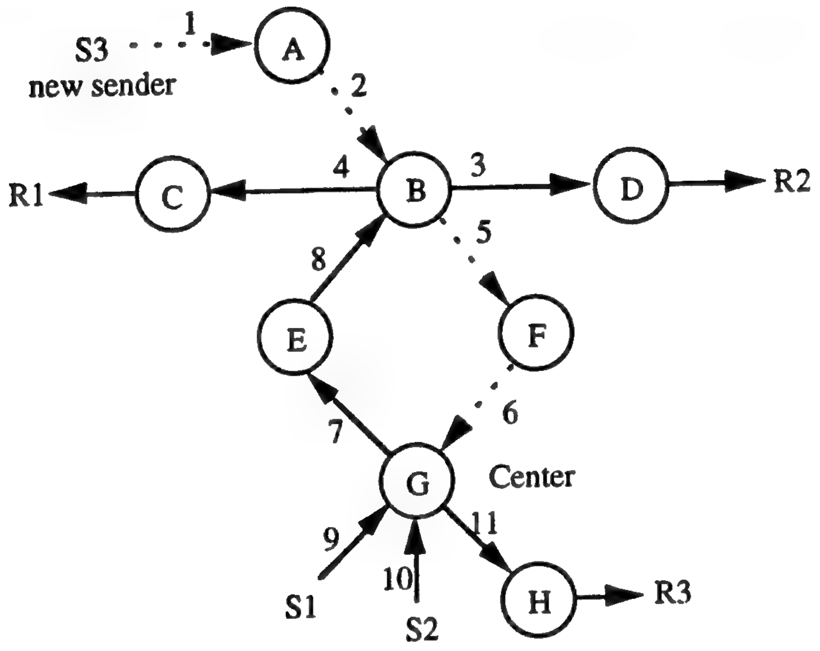


Figure 7. New Sender Example

protocol is illustrated with the following example. Consider the topology in Figure 7. The dashed arrows represent the path of the join-request for a new sender (S3) to the center (router G). Build-State messages are generated at routers B and G and a prune must occur along the path [B, E, G]. Table 1. shows the state of the tree before the join-request, after all Build-State messages have reached their destinations, and after pruning is complete.

Figure 7 illustrates why the Build-State messages must be sent at each on-tree router. If the Build-State messages were sent only when the join-request reached the center, the Build-State message that reached router B would encounter S3 in the source list and the message would not get propagated out interfaces 3 or 4. This is eliminated if Build-State messages are generated at router B as well as the center.

The protocol for adding new receivers is presented in Figure 8. This protocol is illustrated with the following example. Consider the topology in Figure 9. A new receiver

Router	Current State	After Build-State Messages	Pruned State
A	No Senders	S3, 1, {2}	

B	S1, 8, {3, 4}
	S2, 8, {3, 4}
		S1, 8, {3, 4}
		S2, 8, {3, 4}
		S3, 2, {3, 4, 5}	

C	S1, 4, {R1}
	S2, 4, {R1}
		S1, 4, {R1}
		S2, 4, {R1}
		S3, 4, {R1}	

D	S1, 3, {R2}
	S2, 3, {R2}
		S1, 3, {R2}
		S2, 3, {R2}
		S3, 3, {R2}	

E	S1, 7, {8}
	S2, 7, {8}
		S1, 7, {8}
		S2, 7, {8}
		S3, 7, {8}
			S1, 7, {8}
			S2, 7, {8}
			Remove S3

F	No Senders	S3, 5, {6}	

G	S1, 9, {7, 11}
	S2, 10, {7, 11}
		S1, 9, {7, 11}
		S2, 10, {7, 11}
		S3, 6, {7, 11}
			S1, 9, {7, 11}
			S2, 10, {7, 11}
			S3, 6, {11}

H	S1, 11, {R3}
	S2, 11, {R3}
		S1, 11, {R3}
		S2, 11, {R3}
		S3, 11, {R3}	

Table 1. Adding a New Sender

located at router F is to be joined to the tree. The join-request propagates to the center and the join-ACK proceeds to the receiver along the path [A, B, C, D, E, F]. The current state at each router is given in Table 2. As the join-ACK propagates towards the receiver, the state at each router is changed to the New State given in Table 2. The state of the Modified

Sender List (MSL) is also shown in this column. When the join-ACK reaches router D it encounters S1 in the sender list (SL) for that router and S1 is also in the MSL. Thus, for S1 a shorter path to router D exists and S1 must be pruned from the path traversed so far. The state of any router that is different as a result of the prune message is presented in the last column of Table 2.

Some additional functions are required in the join-ACK to accommodate the pruning process. In order to determine the path along which to send the prune message, the join-ACK must record its route as it propagates to the receiver or this path must be discerned by the prune message from the state information at each router (i.e. using the

```
From the center, propagate the join-ACK to the receiver.

1. For any sender at the center with the next hop not in {OGIs}, add the next
hop to the {OGIs}. Add these senders to the Modified Sender List (MSL) in
the join-ACK.

2. Proceed to the Receiver

3. At each hop:
   3.1 If a sender in the MSL is NOT a member of the SL at the hop:
       Add the sender to the SL with the last hop as the incoming interface
       and the next hop as the single OGI.

   3.2 If a sender that is NOT in the MSL is encountered in a SL:
       Check the senders {OGIs} for the next hop:
           If next hop in {OGIs}, do nothing
           If next hop NOT in {OGIs}, add next hop to {OGIs} and add the
               sender to the MSL.

   3.3 If a sender that is in the MSL is encountered in a sender list:
           Prune the previous path of the join-ACK by removing the sender
           from any router for which $|OGI|=1$. When $|OGI| >1$ for that
           sender, or the center is reached, remove the OGI corresponding to
           the next hop for the receiver from the sender's {OGIs}.

           Check the encountered sender's {OGIs}:
               If the next hop $\in$ {OGIs}, remove sender from MSL.
               If the next hop $\notin$ {OGIs}, add next hop to {OGIs}.
```

Figure 8. Protocol to Add New Receivers

incoming and outgoing interfaces from the sender list. When the prune message eventually encounters a set of OGIs of size greater than one, it must determine which interface to prune from this set. Given the identity of the receiver, the interface corresponding to the next hop for the receiver should be the interface that gets pruned. Thus, the prune message must know the sender id that it is pruning and the receiver id it is pruning that sender for.

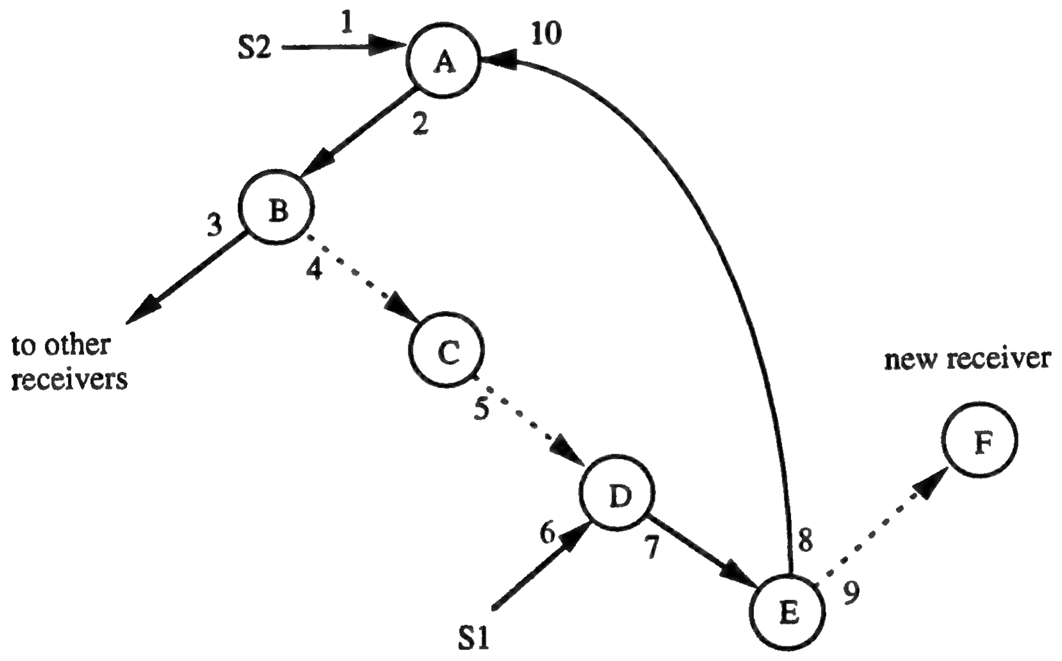


Figure 9. New Receiver Example

The first participant that joins the tree is critical to the correct formation of the tree. If a sender is the first to join, the tree will be built correctly. However, if a receiver joins before any other sender, then there is no state at the center to propagate back towards the receiver. To remedy this, the center can either wait for a sender to join before propagating the join-ACK or the join-ACK can be sent with the center listed as a sender. This second case allows a trail to be built to the receiver that a new sender can follow. No traffic will be sourced from the center and its entry in each SL requires minimal overhead.

Router	Current State	New State	Pruned State

A	S1, 10, {1,2}
	S2, 1, {2}
		No Change
		MSL = {}	

B	S1, 2, {3}
	S2, 2, {3}
		S1, 2, {3, 4}
		S2, 2, {3, 4}
		MSL={S1, S2}
			S1, 2, {3}
			S2, 2, {3,4}

C	No Senders	S1, 4, {5}
		S2, 4, {5}
		MSL={S1, S2}
			Remove S1
			S2, 4, {5}

D	S1, 6, {7}	S1, 6, {7}
		S2, 5, {7}
		MSL={S2}
			Generate Prune

E	S1, 7, {8}	S1, 7, {8, 9}
		S2, 7, {9}
		MSL={S1, S2}	

F	No Senders	S1, 9, {Rec}
		S2, 9, {Rec}	

Table 2. Adding a New Receiver

IV. SIMULATION RESULTS

This chapter describes the evaluation of the center selection and tree construction techniques detailed in the previous chapters. The objective is to compare the tree cost and average path length as the number of concurrent senders is varied for two types of trees. The first is a center-specific shortest path shared tree with the center located according to the proposed approach. The second is a source-specific shortest path tree. The ground work for the following simulations can be found in [23] and the results further support the preliminary findings detailed in that work.

A. ENVIRONMENT

A random network is generated using Waxman's RG1 and RG2 [9] algorithms. Several clusters (meant to simulate separate routing domains) are generated using RG2 and these clusters are connected by links generated using either RG1 or RG2. A maximum cost between any two nodes (L) and the total number of nodes (N) is established. For RG2, every pair of nodes (i,j) is assigned an integer cost ($d_{i,j}$) utilizing a uniform distribution from 1 to L. For RG1, the nodes are scattered in an NxN matrix and the Euclidean distance ($d_{i,j}$) between any node pair (i,j) is calculated. The probability ($p_{i,j}$) that a link exists between a node pair is given by $p_{i,j} = \beta\, e^{-d_{i,j}/(\alpha L)}$. If a link exists, its cost is $d_{i,j}$. If link (i,j) exists then $p_{j,i} = 1$ but the cost $d_{j,i}$ is completely independent of cost $d_{i,j}$. The parameters α and β are defined on the interval (0,1]. A small value of α will cause a relatively greater number of low cost links. The node degree, (λ_i), (the number of links from a given node) for node *i* is approximated by

$$\lambda_i \approx \sum_{j=1, j\neq i}^{N} p_{i,j} = \beta \sum_{j=1, j\neq i}^{N} e^{-d_{i,j}/\alpha L}$$

Since any cost ($d_{i,j}$) inside a cluster is a uniformly distributed integer between [1, L] there will be approximately (N - 1)/L links of each possible value of $d_{i,j}$. This allows the following revision.

$$\lambda_i \approx \frac{\beta(N-1)}{L} \sum_{k=1}^{L} e^{-k/\alpha L}$$

Letting $e^{-1/(\alpha L)} = \rho$ and observing that the above equation is a finite geometric sum of ρ yields

$$\lambda_i \approx \frac{\beta(N-1)\,\rho(1-\rho^L)}{L(1-\rho)}$$

Since λ_i can be approximated by this method for every i, the average node degree λ_{avg} for the entire graph can be approximated by this formula.

$$\lambda_{avg} \approx \frac{\beta(N-1)\,\rho(1-\rho^L)}{L(1-\rho)}$$

Solving for β results in the following

$$\beta \approx \frac{\lambda_{avg}\,L(1-\rho)}{(N-1)\,\rho(1-\rho^L)}$$

If further simplification is desired ρ^L can be approximated as 0 and N - 1 can be approximated as N for L >> 1 and N >> 1 respectively.

The tree cost for each tree is calculated by determining the number of senders that use each link. Let $S_{i,j}$ be the number of senders that use the link from node *i* to node *j* and $d_{i,j}$ be the cost to use that link. Let *ss* be the number of concurrent senders. The tree cost for that link, $tc_{i,j}$, is given by $tc_{i,j} = \min(ss, S_{i,j})d_{i,j}$. The total tree cost, tc_{tot}, is given by

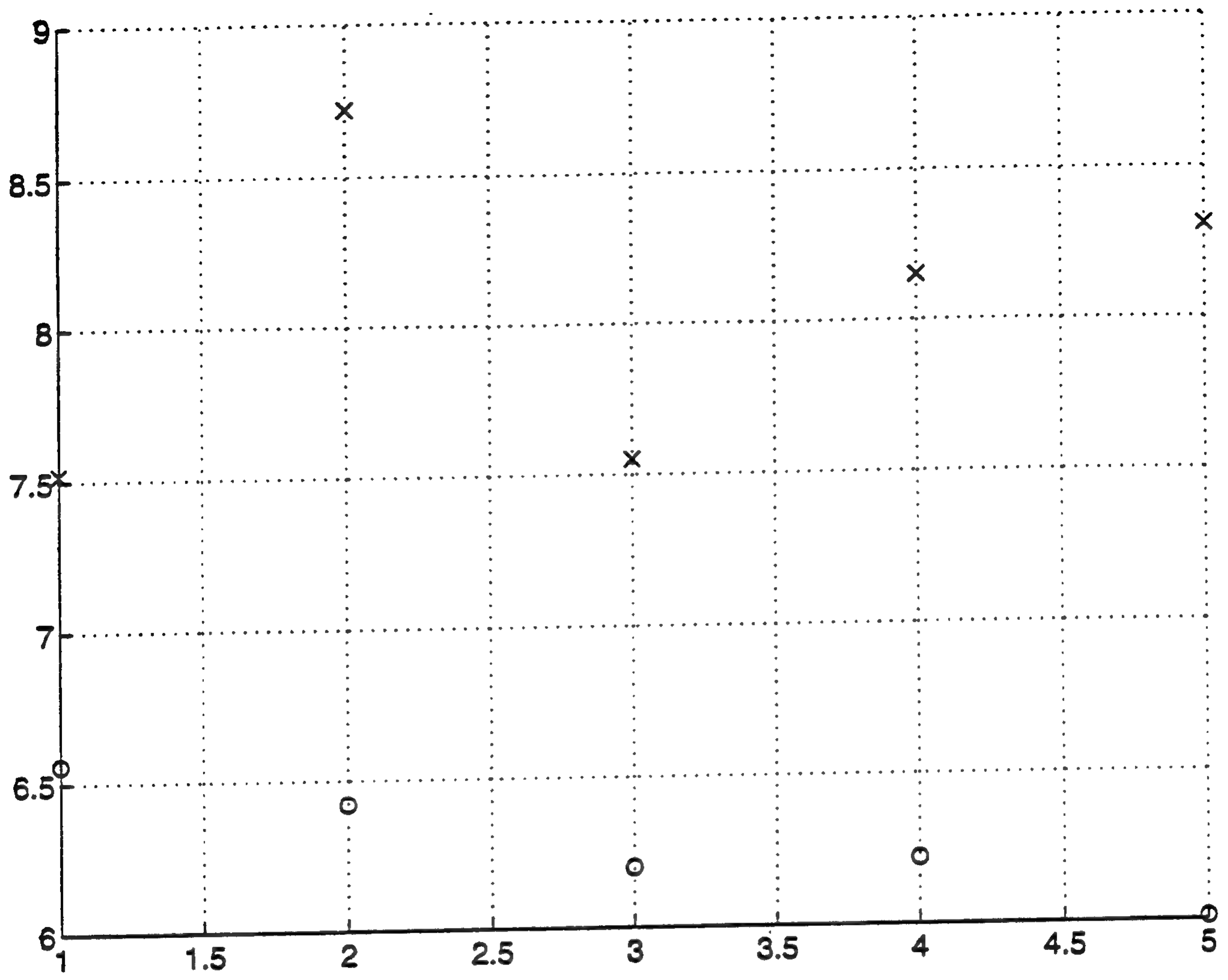
$$tc_{tot} = \sum_{i=1}^{N} \sum_{j=1}^{N} tc_{i,j}$$

For the shared trees, separate send and receive paths are allowed as per the tree construction protocol. For both tree types the average path length is computed as the average number of hops experienced by a sender to all receivers averaged over all senders.

B. RESULTS

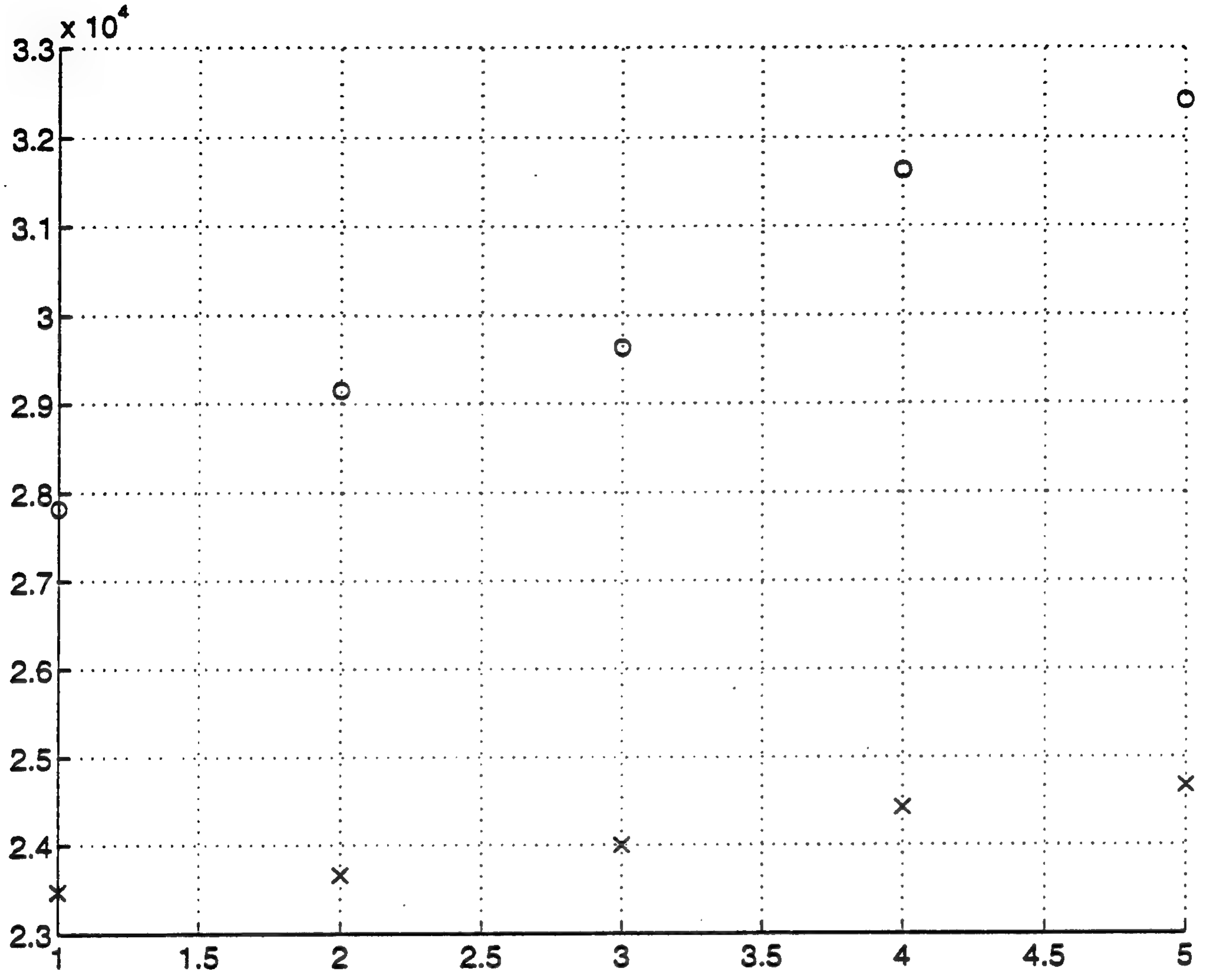
Simulations were carried out to determine how the shared trees compare with source-specific trees as the number of concurrent senders in an interaction increased. The topologies varied from 10-100 nodes over 1 to 10 clusters. The cost of the links between clusters was on average an order of magnitude greater than links within clusters. Node degree was kept in the interval [3,5] by manipulating α and β for each topology. The simulations allowed multiple interactions to be constructed on top of each other. As one interaction was built (each interaction is referred to as an "iteration" of the approach) the network resources required by that interaction were consumed in the state of available bandwidth and additional sessions were then built using the new bandwidth state. Thus, separate bandwidth state was maintained for each tree type to facilitate comparison of the tree types in the presence of multiple interactions.

Figures 10, 11, and 12 show a representative sample of results generated through extensive simulations. Figure 10 shows the average path length (in number of hops) experienced by each sender per iteration. This path length is proportional to the delay the multicast traffic experiences on the tree. Figure 11 shows the tree cost per iteration for a specific number of concurrent senders defined during the simulation. This graph demonstrates how the proposed approach distributes the load over several iterations. Figure 12 shows the tree cost for both trees as the number of concurrent senders is increased (up to the total number of senders defined in the simulation) In Figure 12 the



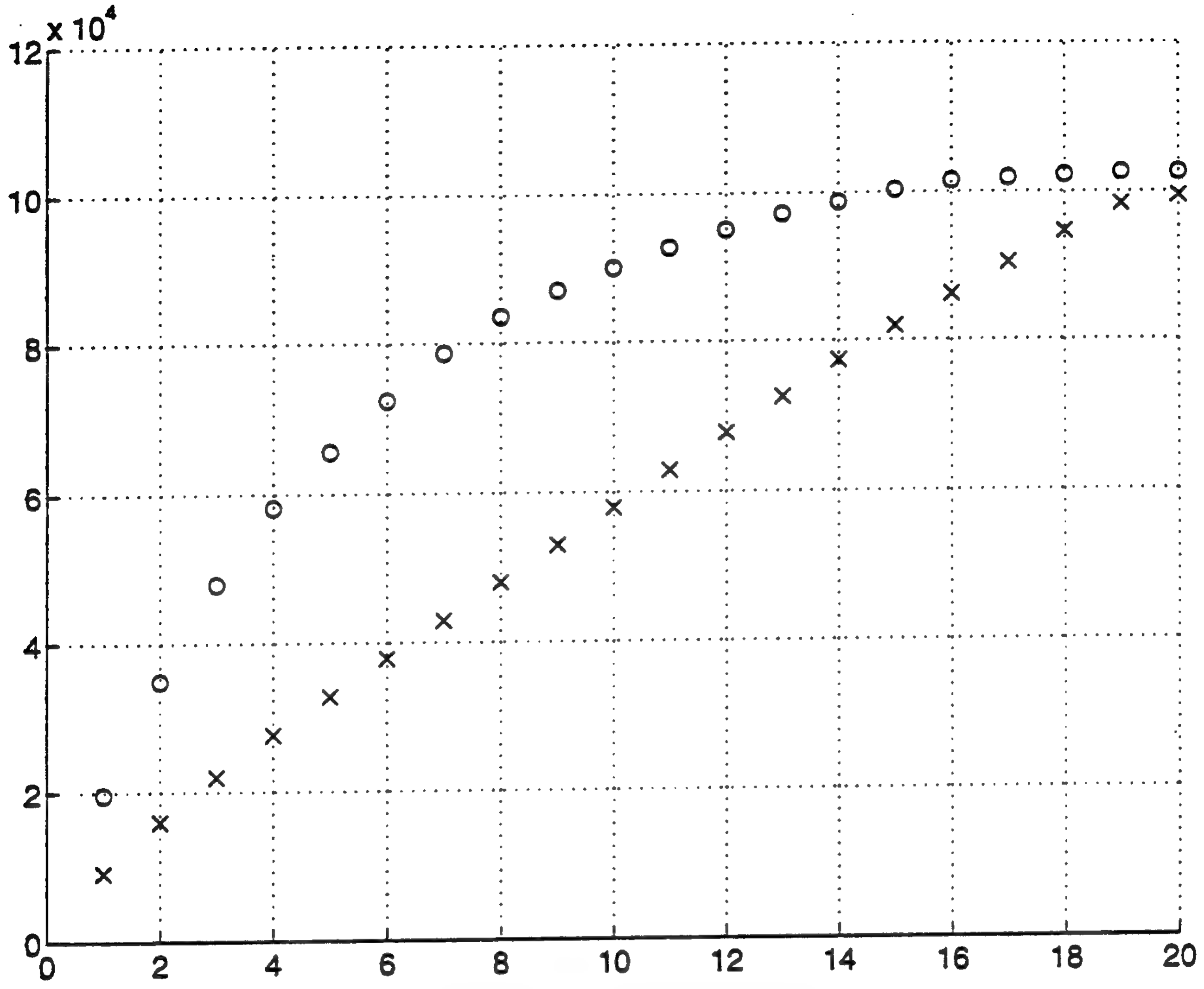
Number of Interactions
x = Shared Tree, o = Source Based Tree

Figure 10. Average Path Length per Interaction



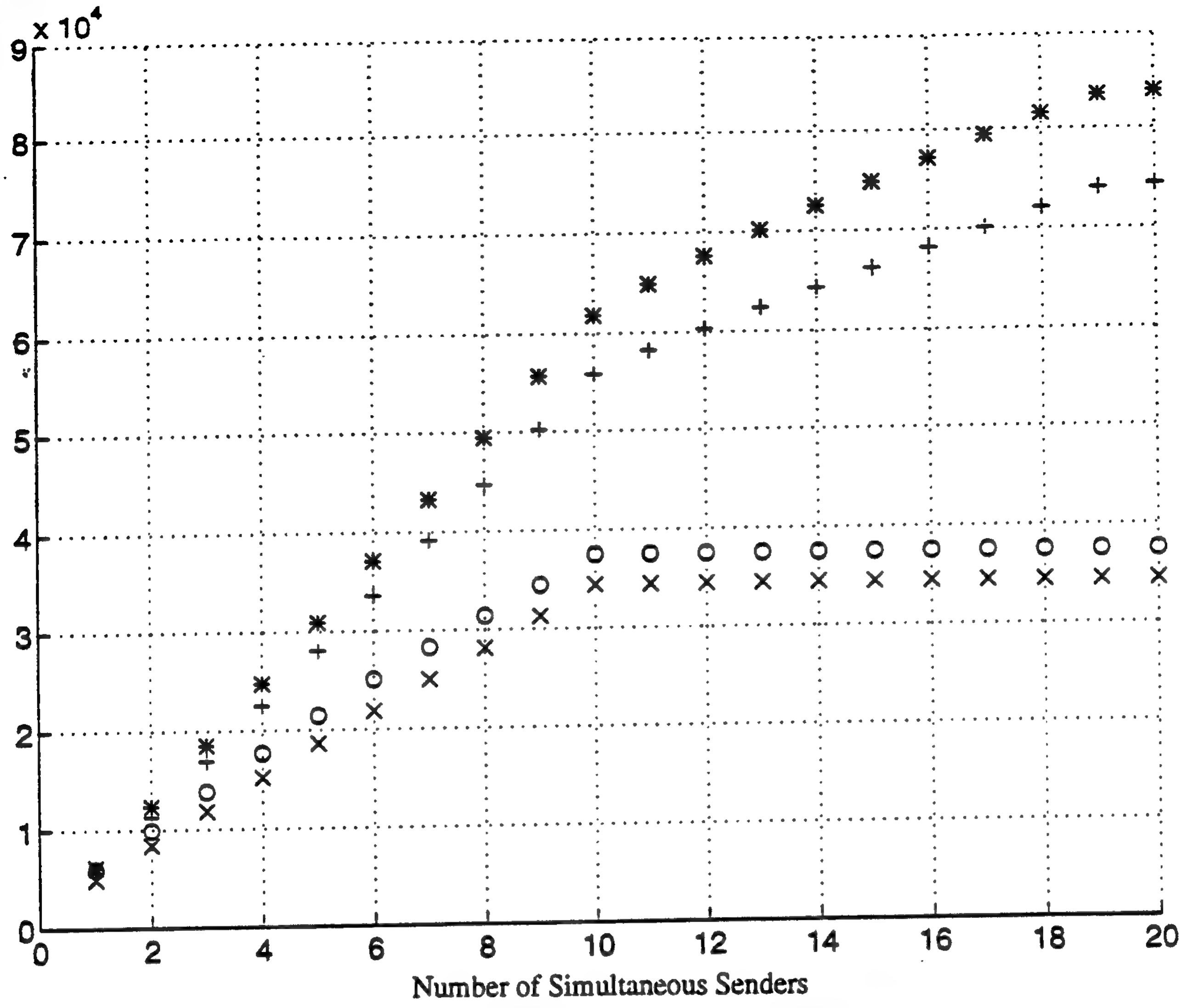
Number of Interactions
x = Shared Tree, o = Source Based Tree

Figure 11. Tree Cost per Interaction



Number of Simultaneous Senders
x = Shared Tree, o = Source Based Tree

Figure 12. Tree Cost vs. Simultaneous Senders on Final Interaction



* = Tree Cost for Receive Tree Rooted at Selected Center
o = Tree Cost for Send Tree Rooted at Selected Center
+ = Tree Cost for Receive Tree Rooted at Least Cost Center
x = Tree Cost for Send Tree Rooted at Least Cost Center

Figure 13. Evaluation of Center Selection Protocol With Respect to the Number of Simultaneous Senders

sender-specific tree starts out with a higher cost, peaks more rapidly, but levels off as the number of concurrent senders increases. The center-specific tree tends to increase at a constant rate. When the number of concurrent senders is large it is possible for the cost of the center-specific tree to exceed that of the sender-specific tree. The shape of the sender-specific tree is due to a greater number of shared links, but these links are shared by only a few senders, whereas the links of the center-specific tree are shared by most senders.

The center selection algorithm was evaluated by comparing the cost of the shared tree against the cost of the "send" and "receive" tree rooted at the optimal send and receive centers. A "send" tree was computed as the one-way tree from all senders to a send center. Likewise, a "receive" tree was computed as the one-way tree from a receive center to all receivers. Optimal send centers were located by computing the least cost from all senders to a router in the network. The costs to that router were averaged over all senders. This average cost was computed for every router in the network and the router representing the least average cost from all senders was selected as the optimal send center. A similar computation was performed to locate the optimal receive center.

Figure 13 shows a plot of the tree cost for the send and receive trees with the one-way tree cost for the proposed shared tree imposed on top of these costs. The costs are shown as the number of concurrent senders is increased. From these tests it is evident that the proposed approach locates centers that represent a good compromise between the sending and receiving requirements of the interaction.

C. SUMMARY

The main conclusion to be drawn from the simulations is that the shared trees do provide a lower tree cost than source-specific trees without sacrificing significantly in delay even for multiple concurrent senders. Source-specific trees are superior to shared trees when the number of concurrent senders is large and when the network is highly interconnected. The cross-over point depends upon both the richness of the topology and the number of concurrent senders.

The simple center location protocol is very effective in generating low cost shared trees. It performs even better in topologies that are not highly interconnected. It must be noted that our results are significant if multiparticipant interactions are to be set up with reservation of resources to guarantee QoS. When reservations are required, each tree cost calculated above can be interpreted as the total number of resources to be reserved in the network to support the corresponding number of concurrent senders.

[illegible]

V. RELATED WORK

In this chapter we examine the extent to which the existing protocols meet the following design goals previously identified for network level multicast with guaranteed QoS. The design goals are:

1. Use of *a priori* information about participants,
2. Automatic location of distribution centers,
3. Flexible selection between source specific and center-specific trees,
4. Tree formation based on resource availability,
5. Support of multiple routing protocols,
6. Minimal tree state information,
7. Minimal per-packet processing in the routers,
8. Participation of senders as well as receivers in reservation.
9. Minimal resource consumption in asymmetric network topologies.

Note, goals 6 and 7 are somewhat conflicting. Table 3 summarizes the comparison among existing protocols. The Distance Vector Multicast Routing Protocol (DVMRP) [5] and Multicast Extensions to OSPF (MOSPF) [7] automatically locate a center since they utilize sender-specific trees only. These protocols are only applied to intra-domain multicast routing and therefore are not required to support multiple protocols. Goal 8 is not listed in the table since none of the existing protocols supports reservations. The completely independent RSVP implements receiver based reservation.

MOSPF is the only existing technique that handles asymmetric network topologies. Since the topological database in MOSPF is stored as a directed graph, the link costs are bi-directional and the Dijkstra shortest path trees are computed using this state information. Techniques using reverse path multicasting (RPM) or some variant of RPM obviously suffer when the link costs are asymmetric [15]. Thus, techniques such as Dense Mode PIM and DVMRP do not support asymmetric topologies. Sparse mode PIM sets up the

Goal	CBT [1]	PIM [2,3,4]	DVMRP [5, 8]	MOSPF [7, 8]	Proposed Approach
Use of *a priori* information	N	N	N	N	Y
Automatic location of distribution centers	N	N	Y	Y	Y
Choice between SST and CST	N	Y	N	N	Y
Tree formation based on resource availability	N	N	N	N	Y
Minimal router processing	Y	Note 1	N	N	Y
Minimal tree state information	Y	Note 2	Y	N	Note 3
Support of multiple routing protocols	N	Y	N	Y	Y
Support for asymmetric topologies	N	N	N	Y	Y

Note 1: Y for sparse mode, N for dense mode
Note 2: N for sparse mode, Y for dense mode
Note 3: Moderate amount of state, less than PIM but more than CBT

Table 3: Existing Approaches vs. Design Goals

packet delivery path as PIM-join messages propagate towards the RP or source for each receiver. Assuming that the path taken by the PIM-join is the shortest path to the RP or source, it may not represent the shortest path that the actual traffic to the receiver must take if the link costs are asymmetric. Thus, the resultant shared or source based trees for Sparse Mode PIM may be sub-optimal. Tree construction for CBT also suffers from a similar phenomenon when the network topology is asymmetric. For ToS based routing the designers of PIM suggest in [2] that a symmetry flag be used by BGP/IDRP [17, 18] that allows PIM to determine if packets will be allowed to travel in the reverse direction. If this flag is not set PIM suggests looking for an SDRP [19] route that has the flag set. This requires SDRP to carry the symmetry flag and that PIM messages follow an SDRP route. This approach appears to select arbitrary paths relative to QoS requirements and then hopes that the paths are symmetric by checking the symmetry flag. Additional questions remain concerning conditions under which the symmetry flag will be set.

The tree state information required at each on-tree router in the approach proposed in this thesis is of the same order as the state information required by MOSPF or Sparse Mode PIM (identified by the multicast forwarding entry for (S,G) required at each on-tree router). So while the proposed approach may not meet goal 6 completely, it meets it to the same extent as some existing techniques.

The proposed approach also compares favorably in size with Sparse Mode PIM. PIM requires that every receiver learn of each sender through an RP. This implies that an RP, must continue to listen to every sender and each sender must continue to send to every RP, even if no receivers are receiving traffic through the RP. Let N_s, N_r, and N_c be the number of senders, receivers, and RPs (in the case of center-specific trees, the number of centers) respectively. In PIM, a new receiver gets all its multicast traffic from a single RP regardless of the number of senders. In the proposed approach each receiver attaches itself to all centers and a sender sends traffic to only one center. A rough estimate of the size of the solution for PIM is $N_s N_c$. This is provided that all receivers get their traffic through an RP. If all receivers opt to gather their traffic from every source, the metric becomes bounded by $N_s N_r + N_s N_c$, a rather significant increase.

PIM requires that the RPs maintain a complete list of senders and that on-tree routers maintain a multicast forwarding entry for the shared tree and for all sources on shortest path trees. The proposed approach requires that a partial list of senders (containing the same information as PIM multicast forwarding entries for (S,G)) be maintained at all on-tree routers. The approach will still scale if the number of senders per CSP and the number of routers shared by more than one tree is small. Some control can be exerted to meet the first condition, but routers near the receivers will likely be members of all trees. If R_i = {routers on-tree for CSP i} and S_i = {senders for CSP i} then a rough estimate for the size of the proposed approach is:

$$\sum_{i=1}^{n} |S_i||R_i|$$

where n is the number of CSPs (equal to N_C in the discussion above).

VI. CONCLUDING REMARKS

This thesis addressed the problem of constructing multicast trees with guaranteed QoS that utilize network resources efficiently. It identified design goals for constructing such trees and presented an integrated approach to achieve an efficient combination of center-specific trees and sender-specific trees based on *a priori* information about participants. It describes a scalable center location mechanism and protocol for locating a distribution center that balances network resource consumption. It describes an approach and protocol for constructing a shared tree around a pre-selected distribution center in a network with asymmetric link costs. It is shown, by simulation modeling, that the resultant center-specific trees are efficient in the presence of multiple concurrent senders in terms of delay and resources consumed.

Since the state information required by the proposed approach introduces some complexity, some alternatives should be examined. The simplest alternative is to force the same path for traffic in both directions. While this will result in the degradation of some traffic (since the path it must traverse is no longer the shortest) some improvement can still be made over existing approaches. A determination can be made to minimize the degradation suffered by the traffic by picking a path that represents the best compromise between the two directions. While the trees in the proposed approach will consume fewer resources, the state information in each router is no longer required. Since the overall goal was to minimize the network resources consumed by the tree (to guarantee QoS), separate send and receive paths are proposed at the expense of minimal tree state information.

The two main contributions of this work were motivated by the idea of proposing enhancements to existing protocols. The center selection mechanism was motivated by the need to determine a location for cores in a CBT approach or RP's in a Sparse Mode PIM approach. The proposed tree construction algorithm has its origins in the tree construction phase proposed by CBT with the state information taken from Sparse Mode PIM and MOSPF. While the need for the state information in the proposed approach (separate paths for send and receive traffic) is different than that of Sparse Mode PIM (the superposition of shared and source-based trees) the complexity imposed by each remains the same.

The following issues need to be addressed if an implementation of this approach is to be undertaken:

1. A detailed specification of the tree construction protocol, particularly at the boundary conditions,

2. A detailed specification of a protocol to remove state information for departing participants. (We expect this protocol to be similar to the tree construction protocol),

3. Analysis of the tree construction protocol under transient conditions (e.g. during simultaneous joins),

4. Formal specifications of the above protocols and associated proofs,

5. Detailed specifications of the registration protocol, and reservation protocol,

6. A method for reconfiguring the distribution centers over the lifetime of an interaction.

The contributions of this work are:

1. Use of *a priori* information about participants to provide multicast data distribution that permits a flexible combination of center-specific and sender-specific trees,

2. An approach to algorithmically locate a center for a center-specific tree,

3. An approach to center-specific tree construction in the presence of network asymmetry,

4. Simulations detailing the quality of the resultant trees.

APPENDIX A

PROGRAM CODE

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                             TREE_ANALYZER.M                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% Version 6 September 5 1994                                            %
% Eric Klinker                                                          %
%                                                                       %
% Removed the 3 loops that converted 0 entries in cost and bw_rsvd to  %
% Inf entries. This is now done with version 3 of function get_cost.   %
%                                                                       %
% Version 5 August 16, 1994                                             %
% Eric Klinker                                                          %
%                                                                       %
% This version does not compute Steiner trees but instead evaluates the%
% findcore center against a send and receive center that is selected   %
% as representing the node with the least average cost to all senders  %
% receivers.                                                            %
%                                                                       %
% This version also removes the reverse path tree that was in earlier  %
% versions                                                              %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 4 August 2, 1994
% This version performs all calculations on the three trees one at a
% time. This allows one copy of hops to do all the tasks.
% This file was modified by Eric Klinker 30 July 1994
% It is the same as thesisgraph2 except that it calls the new Steiner
% Tree approximation Steiner2 just after calling Findcore2.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The graph generator portion of this program is based on RG2 as
% described by W. M. Waxman in his article "Routing of Multipoint
% Connections" in IEEE Journal on Selected Areas in Communications,
% December 1988.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eric B. Boyer November 1993
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Modified by Eric Klinker July 1994
%
% Made the adjacency matrix d_cor global in order to build the matrix
% in the function FINDCENT.
%
% Made adj_sprs global as well even though it is returned by steiner2
% 31 July 1994
```

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%

seed=input('Input a new random number seed for this run -->');
rand ('seed',seed);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% xcor and ycor are dimensions for locating the nodes
% n = number of nodes to be represented
% alpha and beta are parameters of edge descriptions
% l (el) is the max distance between any pair of nodes.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

global sp cost_cor cost_sh cost_sprs n hops cps
global lambda_within lambda_btwn nodes_pc basecost_within
global basecost_btwn root
global numclusters l_within l_btwn bw_this_iteration bw_rsvd_cor
global bw_rsvd_sh
global bw_rsvd_sprs maxlength_btwn maxlength_within gridarea_btwn
global adj_cor adj_sprs cor_tree_hops sprs_tree_hops
% added for new get_cps1 (senders and receivers)
global senders receivers bw_rsvd_rp cost_rp

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The following command starts the script file that asks the user for
% the parameters alpha, beta, n, and l and determines the values of
% xcor and ycor.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


get_vals


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The next command is a script file that determines the existence
% of edges between any two nodes and the cost to use that edge. It
% generates the matrix "bw[n,n]" that is a table that contains the
% bandwidth reserved between any two nodes.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

edges

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The next command is a script file that plots the graph. The values
% required for the plot are generated in edges.m.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure

plt_grph
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Some description of the various matrices is in order. There are
% several matrices associated with each tree. The first is the bw_rsvd
% matrix which is the adjacency matrix computed in edges. This matrix
% describes the connectivity of the entire network and the reserved
% bandwidth on any link. If a separate cost function is used the cost
% matrix for the entire topology is generated with a call to get_cost.
%
% bw_rsvd_cor describes the entire topology adjacency that has been
% updated with the presence of one or more core based trees. When a tree
% built the bandwidth required by that tree is added to the current
% state of the network. This is done in bw_rsvd_cor. It does not
% describe the adjacencies of the core based tree. Likewise bw_rsvd_sh
% is the bw state for sender based trees. bw_rsvd_spars is the bw
% state for steiner trees.
%
% The adjacency matrices describe which links are being utilized by the
% current tree. With no cost function, the matrix describes the reserved
% bandwidth. adj_cor, and adj_sprs are the two adjacency matrices.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

bw_rsvd_cor=bw_rsvd;
bw_rsvd_sh=bw_rsvd;
bw_rsvd_sprs=bw_rsvd;
bw_rsvd_rp = bw_rsvd;

iterations = input(' How many interactions? -->');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The next command is a script file that asks for the number of
% critical participants and randomly locates them among the nodes. The
% node address of each cp is kept in the vectors senders and receivers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_cps
NumSend = length(senders);
NumRec = length(receivers);


for loop=1:iterations

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Convert the bw_rsvd matrix into a valid cost matrix based on
% the parameters base_cost and lambda
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      cost_cor=get_cost(bw_rsvd_cor);
      cost_sh=get_cost(bw_rsvd_sh);
      cost_sprs=get_cost(bw_rsvd_sprs);
      cost_rp = get_cost(bw_rsvd_rp);
      bw_this_iteration = ceil (rand*l_within/25);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The following command initializes 4 matrices. sp is the distance to
% the column node on the shortest path tree sourced at the row node,
% stein is the distance to the column node with the row node being the
% source of the steiner tree. hops and hopstein are the intermediate
% nodes between the source and destination node for sp and stein
% respectively.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      hops = zeros(n*n , n-2);
      sp = -ones(n, n);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The script file SBT performs all the calculations for the source
% based trees using all of the source based data (bw_rsvd_sh, cost_sh,
% sh_cnt) and produces the average path length for the shortest path
% tree total and on a per sender basis, as well as the tree cost
% (shcst, sh_avg)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      SBT
      fprintf('.')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The script Shared_Tree performs all the calculations for the shared
% tree proposed in this thesis. It uses all of the core data
% (bw_rsvd_cor, cost_cor) to build the shared tree and produces cor
% variables as per the SBT script above.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      Shared_Tree
      fprintf('.\n')


end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Evaluate the core by calculating the Best Send Center and Best Receive
% center and for send and receive trees. Then compare the half tree cost
% for the Shared_Tree against these two trees. SendCent and RecCent
% locate the two centers and Eval_Core performs the evaluation.
%
% This can be done here because all the calculations are performed with
% cost_cor (and not rsvd_bw_cor which has been updated) and we are only
% concerned with the result after the last iteration.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


Core_Eval
```

```
% Plot the data.

out_data
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              BUILD_SHARED.M                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This script builds the shortest path shared tree around the core
% selected by the script findcore.
%
% The result is the adjacency matrix adj_cor which describes the tree.
%
% This version allows separate send and receive shortest paths to the
% tree. Functionally this would result in routing loops unless some
% means for control (such as source lists) at each router is employed.
% This code simply evaluates the quality of a tree so this problem is
% ignored here.
%
% It is called from the script Shared_Tree.m
%
% Version 1 August 3, 1994
% Eric Klinker
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Record the path from all senders to the core

for i = 1: NumSend
      hopcnt = last_hop(hops((senders(i)-1)*n+core,:));
      hopsrc = senders(i);
      for k = 1:hopcnt+1
            if k == hopcnt+1
                  hopdst = core;
            else
                  hopdst = hops((senders(i)-1)*n+core,k);
            end
            if hopsrc ~= hopdst
                  adj_cor(hopsrc,hopdst) = bw_rsvd_cor(hopsrc,hopdst);
            end
            hopsrc = hopdst;
      end
end

% Record the path from the core to all senders

for j = 1:NumRec
      hopcnt = last_hop(hops((core-1)*n+receivers(j),:));
      hopsrc = core;
      for k = 1:hopcnt+1
            if k == hopcnt+1
                  hopdst = receivers(j);
            else
                  hopdst = hops((core-1)*n+receivers(j),k);
            end
            if hopsrc ~= hopdst
                  adj_cor(hopsrc,hopdst) = bw_rsvd_cor(hopsrc,hopdst);
            end
            hopsrc = hopdst;
        end
    end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              Core_Eval.m                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 1 August 16,1994
% Eric Klinker
%
% This script performs the necessary calculations to evaluate the core
% selected by findcore against SendCent and RecCent.
%
% The script is very similar to the performance calculations performed
% in SendCent and RecCent except in each case "core" is used instead of
% sendcent or reccent.
%
% Version 2 August 20, 1994
% Eric Klinker
%
% The script changed the evaluation of both receive trees (RecCore and
% RecCent). The change involved counting the number of senders that
% must use each link from the core to each receiver and then using
% the smaller of that number or the number of simultaneous senders
% to calculate a tree cost.
%
% The tree cost will no longer be linear as the simultaneous senders
% increases.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Perform evaluation of core first, using current state of hops
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% the following segment counts the number of senders that use each link
% to get to sendcent.

cor_send_cnt = zeros(n,n);
cor_sendcst = zeros(1,NumSend);
tree_cost = zeros(n,n);

for i = 1:NumSend
      hopsrc = senders(i);
      hopcnt = last_hop(hops((senders(i)-1)*n+core,:));
      for k = 1:hopcnt+1
            if k == hopcnt+1
                  hopdst =core;
            else
                  hopdst = hops((senders(i)-1)*n+core,k);
            end

      cor_send_cnt(hopsrc, hopdst) = cor_send_cnt(hopsrc,hopdst)+1;
      hopsrc = hopdst;
      end
end
```

```
% Calculate the tree cost for any number of concurrent senders



for k = 1:NumSend
      for i = 1:n
            for j = 1:n
                  tree_cost(i,j) = Inf2zero(cost_cor(i,j))*...
                  min(cor_send_cnt(i,j),k);
            end
      end

      cor_sendcst(k) = sum(sum(tree_cost));

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Perform a link count to be used in the evaluation. This requires that
% the RecCent entries in hops be updated using the new adjacency matrix
% This should not be computationally intensive since there is only one
% sender (RecCent).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% the following segment counts the number of links required to get to
% each receiver.

cor_rec_cnt = zeros(n,n);
cor_reccst = zeros(1,NumSend);

for i = 1:NumRec
      hopsrc = core;
      hopcnt = last_hop(hops((core-1)*n+receivers(i), :));
      for k = 1:hopcnt+1;
            if k == hopcnt+1
                  hopdst = receivers(i);
            else
                  hopdst = hops((core-1)*n + receivers(i), k);
            end

            cor_rec_cnt(hopsrc,hopdst) = ...
            cor_cnt(hopsrc,hopdst);
      end
      hopsrc = hopdst;
end


for k = 1:NumSend %for any number of concurrent senders
      for i = 1:n
            for j = 1:n
```

```
                        tree_cost(i,j) = Inf2zero(cost_cor(i,j))...
                           *min(cor_rec_cnt(i,j),k);
                  end
            end

            cor_reccst(k) = sum(sum(tree_cost));
      end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Perform calculations for sendcent next
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Perform a link count for use in the evaluation. This requires that
% hops be updated using the new adjacency matrix. This should not be
% computationally intensive as there is only one receiver (sendcent).


for i = 1:NumSend
      [sp(senders(i),:), hops((senders(i)-1)*n+1 : senders(i)*n, :)] ...
            = shrtpath(senders(i), adj_send);
end


% The following segment counts the number of senders that use
% each link to get to sendcent.

send_cnt = zeros(n,n);
sendcst = zeros(1, NumSend);

for i = 1:NumSend
      hopsrc = senders(i);
      hopcnt = last_hop(hops((senders(i)-1)*n+sendcent,:));
      for k = 1:hopcnt+1
            if k == hopcnt+1
                  hopdst =sendcent;
            else
                  hopdst = hops((senders(i)-1)*n+sendcent,k);
            end

            send_cnt(hopsrc, hopdst) = send_cnt(hopsrc,hopdst)+1;
            hopsrc = hopdst;
      end
end

% Calculate the tree cost for any number of concurrent senders


for k = 1:NumSend
      for i = 1:n
            for j = 1:n
                  tree_cost(i,j) = Inf2zero(cost_cor(i,j))...
                        *min(send_cnt(i,j),k);
            end
```

```
        end

        sendcst(k) = sum(sum(tree_cost));

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Perform calculations for "reccent" next
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Perform a link count to be used in the evaluation. This requires that
% the RecCent entries in hops be updated using the new adjacency matrix.
% This should not be computationally intensive since there is only one
% sender (RecCent).


for i = 1:NumSend
      [sp(senders(i),:), hops((senders(i)-1)*n+1 : senders(i)*n, :)]...
            = shrtpath(senders(i), adj_rec);
end



% The following segment counts the number of links required to get to
% each receiver.

rec_cnt = zeros(n,n);
rec_cost = zeros(n,n);
reccst = zeros(1,NumSend);

for i = 1:NumRec
      hopsrc = reccent;
      hopcnt = last_hop(hops((reccent-1)*n+receivers(i), :));
      for k = 1:hopcnt+1;
            if k == hopcnt+1
                  hopdst = receivers(i);
            else
                  hopdst = hops((reccent-1)*n + receivers(i), k);
            end

            if rec_cnt(hopsrc,hopdst) == 0
                  rec_cnt(hopsrc,hopdst) = ...
                        cor_cnt(hopsrc,hopdst);
            end

            hopsrc = hopdst;
      end
end


for k = 1:NumSend %for any number of concurrent senders
      for i = 1:n
            for j = 1:n
                  tree_cost(i,j) = Inf2zero(cost_cor(i,j))...
```

```
                                        *min(rec_cnt(i,j),k);
                        end
                end

                reccst(k) = sum(sum(tree_cost));
        end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              Inf2zero                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function is used in thesisgraph.m. Its primary use is to change
% the Inf in a adjacency matrix modified by tree.m to zeros so that the
% cost of the tree can be determined.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% out = Inf2zero(in) where in is a matrix of any size and
% out is the same input matrix but with any Inf's changed to 0.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = Inf2zero(in)

[row, col] = size(in);

for i = 1:row
      for j = 1:col
            if in(i,j) == Inf
                  out(i,j) = 0;
            else
                  out(i,j) = in(i,j);
            end
      end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              RecCent.m                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 1 August 16, 1994
% Eric Klinker
%
% This script will locate a receive center (reccent) to evaluate the
% core determined in findcore against. This center will represent
% the node that has the least average cost from the node to all
% receivers.
%
% The adjacency matrix contains only the links that are on
% the shortest paths from the center to all receivers.
%
% The core evaluation is performed in another script (Core_Eval.m)
%
% Version 2 August 20, 1994
% Eric Klinker
%
% This version added the paths that the senders would have to traverse
% to get to the receive center. This was necessary for version 2 of
% Core_Eval which examines the number of senders that use each link
% to the receiver.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

node_cost = zeros(1,n);
rec_sp = Inf2zero(sp(:, receivers));
adj_rec = Inf*ones(n,n);

for i = 1:n
      node_cost(i) = (sum(Inf2zero(rec_sp(i,:))))/length(receivers);
end

[m, reccent] = min(node_cost);

% Build the tree from the reccent to all receivers

for i = 1:NumRec
      hopcnt = last_hop(hops((reccent-1)*n+receivers(i),:));
      hopsrc = reccent;
      for k = 1:hopcnt+1
            if k == hopcnt+1
                  hopdst = receivers(i);
            else
                  hopdst = hops((reccent-1)*n+receivers(i),k);
            end

            if hopsrc ~= hopdst
                  adj_rec(hopsrc,hopdst) = cost_cor(hopsrc, hopdst);
            end
            hopsrc = hopdst;
      end
```

```
end

% Connect the senders to the reccent that links can later be
% counted

for i = 1:NumSend
      hopcnt = last_hop(hops((senders(i)-1)*n+reccent,:));
      hopsrc = senders(i);
      for k = 1:hopcnt+1;
            if k == hopcnt+1
                  hopdst = sendcent;
            else
                  hopdst = hops((senders(i)-1)*n+reccent, k);
            end

            if hopsrc ~= hopdst
                  adj_rec(hopsrc,hopdst) = cost_cor(hopsrc,hopdst);
            end

            hopsrc = hopdst;
      end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                               SBT.M                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 1 August 2, 1994
% Eric Klinker
%
% Script called from thesisgraph.m to perform calculations for the
% Source based trees.
%
% The script first calculates the sp matrix and hops matrix based
% on the cost_sh matrix.
%
% The script then uses link_use to calculate the tree cost and
% average path length.
%
% Version 2 August 4, 1994
% This modification adjusts the state of the network bandwidth at
% the end of all other calculations. This new bandwidth state is
% used in the next session
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tree_cost = zeros(n,n);

% Calculate hops

for i=1:n
      [sp(i,:), hops((i-1)*n+1 : i*n, :)] = shrtpath(i, cost_sh);
end


% Get a link count and average path length through link_use
% then calculate the tree cost.

[sh_cnt, sh_sender_pathlen, sh_avg(loop)] = link_use;



for k = 1:length(senders)  %for any number of concurrent senders

      for i = 1:n
            for j = 1:n
                  tree_cost(i,j) = Inf2zero(bw_rsvd_sh(i,j))...
                        *min(sh_cnt(i,j),k);
            end
      end

      shcst(k) = sum(sum(tree_cost));
end

sh_conc_send(loop) = shcst(conc_send);

% Adjust the network state, each sender consumes 1% of
% the max intra-domain link cost (5 units)

for i = 1:n
      for j = 1:n
            bw_rsvd_sh(i,j) = bw_rsvd_sh(i,j) + ...
                  (min(sh_cnt(i,j), conc_send)*5);
      end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                            SendCent.m                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 1 August 16, 1994
% Eric Klinker
%
% This script will locate a send center to evaluate the core determined
% in findcore against. This center (sendcent) will represent the node
% that has the least average cost from all senders.
%
% The adjacency matrix contains only the links that are on the shortest
% paths from each sender to the sendcent.
%
% The core evaluation is performed in another script (core_eval.m)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Calculate the average cost from all senders to each node and select
% the node with the minimum average cost as the send center.

node_cost = zeros(1,n);
send_sp = Inf2zero(sp(senders,:));
adj_send = Inf*ones(n,n);

for i = 1:n
      node_cost(i) = sum(Inf2zero(send_sp(:,i)))/length(senders);
end

[m, sendcent] = min(node_cost);


% Build the tree from all the senders to the sendcent

for i = 1:NumSend
      hopcnt = last_hop(hops((senders(i)-1)*n+sendcent,:));
      hopsrc = senders(i);
      for k = 1:hopcnt+1;
            if k == hopcnt+1
                  hopdst = sendcent;
            else
                  hopdst = hops((senders(i)-1)*n+sendcent, k);
            end

            if hopsrc ~= hopdst
                  adj_send(hopsrc,hopdst) = cost_cor(hopsrc,hopdst);
            end

            hopsrc = hopdst;
      end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              Shared_Tree.m                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 1 August 2, 1994
% Eric Klinker
%
% This script builds a shared tree based on the algorithm proposed
% in this thesis. Then the average path length and tree cost are
% calculated.
%
% The hops matrix is first calculated for the shared tree data.
% From this matrix and the sp matrix the tree is built with the
% script findcore. Then using link_use the tree cost and path length
% are calculated.
%
% Version 2 August 4, 1994
% This version updates the network bw state after all calculations.
% This new state is used when building the next session. Each sender
% currently consumes 5 units of BW which represents 1% of the max
% intra-domain link cost, set as a default. If max cost changes,
% the senders still reserve the same amount, but later versions may
% update the bw consumption with this value.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tree_cost = zeros(n,n);

% Re-Calculate hops and sp with the shared tree cost matrix.

for i = 1:n
      [sp(i,:), hops((i-1)*n+1 : i*n, :)] = shrtpath(i, cost_cor);
end


adj_cor = Inf*ones(n,n);  %adjacency matrix that describes the tree.

findcore

% Build the tree around the core selected in findcore above

Build_Shared

% Locate SendCent and RecCent to evaluate the core against. The
% adjacency matrices for the these two counters are adj_send
% and adj_rec. These calculations require the current state of sp.

SendCent
RecCent


for i = 1:NumSend
      [sp(senders(i),:), hops((senders(i)-1)*n+1 : senders(i)*n, :)]...
            = shrtpath(senders(i), adj_cor);
```

```
end


[cor_cnt, cor_sender_pathlen, cor_avg(loop)] = link_use;


for k = 1:length(senders)  %for any number of concurrent senders

      for i = 1:n
            for j = 1:n
                  tree_cost(i,j) = Inf2zero(bw_rsvd_cor(i,j))...
                        *min(cor_cnt(i,j),k);
            end
      end

      crcst(k) = sum(sum(tree_cost));
end

cor_conc_send(loop) = crcst(conc_send);


% Adjust the state of reserved bandwidth.
% Each sender requires 1% of the max intra domain link
% cost .01(500) = 5 units.

for i = 1:n
      for j = 1:n
            bw_rsvd_cor(i,j) = bw_rsvd_cor(i,j) + ...
                  (min(cor_cnt(i,j), conc_send)*5);
      end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                             ZERO2INF.M                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function is used in findcore.m to change zero entries of a
% shortest path matrix to Infs. This allows subsequent routines to
% select the minimum value in the matrix without selecting cost to
% itself (zero).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% out = Zero2Inf(in) where in is a matrix of any size and out is the
% same input matrix with 0's changed to Inf
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eric Klinker 30 July 1994
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = Zero2Inf(in)

[row, col] = size(in);

for i = 1:row
      for j = 1:col
            if in(i,j) == 0
                  out(i,j) = Inf;
            else
                  out(i,j) = in(i,j);
            end
      end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                               EDGES.M                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This script file is part of thesisgraph.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% It determines the edges and their cost between nodes and generates
% information required for the plot of the graph.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% John W. Eichelberger March 1994
%
% Variables:
% nodes_pc - row of number of nodes per cluster
% dist_btwn_nodes,dbn_copy - a random generated distance matrix,
% used to simulate RG2 node distribution within a cluster
% root,root_copy - matrix of connector nodes btwn clusters
% rootlist - column of all root connector nodes
% bw_rsvd - initial conditions of reserved bandwidth on the ntwrk
%
% Algorithm:
% - generate random number of nodes per cluster
% - generate distance btwn nodes within a cluster for RG2 dist.
% - generate a root in each cluster to connect with other
% clusters
% - generate grid values of root connectors to simulate RG1
% distribution
% - calculate dist_btwn_node values for btwn clusters based on
% user selected distribution
% - using probability test, determine which edges are valid
% - since clusters may have been isolated, check and add random
% edges btwn clusters, if necessary.
% - check for isolated groups of clusters and reconnect to ntwrk
% - since nodes within a cluster may have been isolated, check
% and add random edges within clusters, if necessary
% - generate initial bw_rsvd based on final dist_btwn nodes mtrx
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Asymmetric loading added by Eric Klinker July 1994
%
% To add asymmetry to the load on the network we assumed several
% things.  We keep the distance between nodes
% (the matrix dist_btwn_nodes) symmetric but instead change the
% bandwidth reserved matrix at the end of this program.
% This required a change to the way bandwidth is stored.
%
% Before the bw_rsvd matrix was filled with the distance
% between the nodes divided by the maxlength between clusters
% multiplied by the
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
dist_btwn_nodes = zeros(n,n);
nodes_pc=zeros (1,numclusters);
bw_rsvd=zeros(n,n);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate RG2 values for within clusters, regardless of connector node
% All nodes are in a cluster or singularly defined as clusters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


loc_total=n;

for k=1:numclusters-1
      nodes_pc(1,k)= ceil (rand*(loc_total)); 1
      oc_total=loc_total-nodes_pc(1,k);
      if loc_total < numclusters - k
            difference= numclusters-k-loc_total;
            loc_total=loc_total + difference;
            nodes_pc(1,k)=nodes_pc(1,k)-difference;
      end
end

nodes_pc(1,k+1)=loc_total;

loc_total = 0;
for k=1:numclusters

      for i= loc_total +1:nodes_pc(1,k)+loc_total-1
            for j=i+1:nodes_pc(1,k)+loc_total
                  dist_btwn_nodes(i,j) = rand*maxlength_within;
                  dist_btwn_nodes(j,i) = dist_btwn_nodes(i,j);
            end
      end

      loc_total=loc_total+nodes_pc(1,k);
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Determine connector node matrix for between clusters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

loc_total=0;
root = zeros(numclusters,numclusters);
for i=1:numclusters
      for j=1:numclusters
            if i==j
                  root(i,j)=0; else
                  root(i,j)=ceil(rand*nodes_pc(1,i))+loc_total;
            end
      end

      loc_total = loc_total+nodes_pc(1,i);
```

```
end

root_copy = root;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get grid positions for RG1 and RG3 calculations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

temp=length(nonzeros(root));
if rg_factor==1 | rg_factor==3
      posit= zeros(temp,3);
      root_list=nonzeros(root);
      for k=1:temp
            if any (posit(:,1) == root_list(k))
                  ;
            else
                  posit (k,1) = root_list(k);
                  posit (k,2) = ceil (rand*sqrt(gridarea_btwn));
                  posit (k,3) = ceil (rand*sqrt(gridarea_btwn));
            end

      root_list(k)=0;
      end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate values for between clusters based on user input
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for i=1:numclusters-1
      for j=i+1:numclusters
            if rg_factor==1 | rg_factor==3
                  [temp,temp2]= find (posit(:,1)==root(i,j));
                  [temp3,temp4]=find (posit(:,1)==root(j,i));
            end

            if rg_factor==1 % RG1 is selected %
            dist_btwn_nodes(root(i,j),root(j,i)) =...
               ceil(sqrt((posit(temp,2)-posit(temp3,2))...
               ^2 + (posit(temp,3)-posit(temp3,3))^2)+.01);
            end

            if rg_factor==2
            dist_btwn_nodes(root(i,j),root(j,i)) = ...
               ceil(rand*maxlength_btwn+.01);
            end

            if rg_factor==3 % 50/50 is selected %
                  if rand > .5
                        dist_btwn_nodes(root(i,j),root(j,i))=...
                           ceil(sqrt((posit(temp,2)-posit(temp3,2))...
                           ^2 + (posit(temp,3)-posit(temp3,3))^2)+.01);
                  else
                        dist_btwn_nodes(root(i,j),root(j,i)) = ...
```

```
                              ceil(rand*maxlength_btwn+.01);
                        end
                  end

                  dist_btwn_nodes(root(j,i),root(i,j))=...
                     dist_btwn_nodes(root(i,j),root(j,i));
            end
      end

      dbn_copy = dist_btwn_nodes;

      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      % Determine between-cluster edges using upper triangular matrix
      % bc determines which edges are valid
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      for i=1:numclusters-1
            for j=i+1:numclusters

                  if (rand >beta_btwn*exp(-dist_btwn_nodes(i,j)/...
                     (maxlength_btwn*alpha_btwn)))
                        dist_btwn_nodes(root(j,i),root(i,j))= 0;
                        dist_btwn_nodes(root(i,j),root(j,i))= 0;
                        root(i,j)= 0; root(j,i)=0;
                  end

            end
      end

      for i=1:numclusters
            vec(1,i) = nnz(root(:,i));
      end

      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      %Make sure of no isolated clusters
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      for k=1:numclusters
            if (vec(:,k)==0)
                  temp=ceil(rand*numclusters);

                  while temp==k
                        temp=ceil(rand*numclusters);
                  end

                  root(k,temp) = root_copy(k,temp);
                  root(temp,k) = root_copy(temp,k);
                  dist_btwn_nodes(root(k,temp),root(temp,k))=...
                     dbn_copy(root(k,temp),root(temp,k));
                  dist_btwn_nodes(root(temp,k),root(k,temp))=...
                     dbn_copy(root(temp,k),root(k,temp));

            end
      end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Ensure no isolated fragments of clusters%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

not_conn=[1:numclusters];
tree_list=[];
while length(tree_list) ~= numclusters
      conn_nodes=not_conn(1);
      count=1;

      while length(conn_nodes) >= count
            temp= find(root(conn_nodes(count),:)~=0);

            for m=1:length(temp)
                  if any(conn_nodes==temp(m))
                        temp(m)=0;
                  end
            end

            if nonzeros(temp) ~= []
                  conn_nodes=[conn_nodes;nonzeros(temp)];
            end

            count=count+1;
      end

      if length(conn_nodes)~= length(not_conn)
            not_conn=[];
            for i=1:numclusters
                  if any(conn_nodes == i)
                        ;
                  elseif not_conn==[]
                        not_conn=i;
                  else
                        not_conn=[not_conn;i];
                  end
            end
      end

      if tree_list ~= []
            i=tree_list(ceil(rand*length(tree_list)));
            j=conn_nodes(ceil(rand*length(conn_nodes)));
            root(i,j)=root_copy(i,j);
            root(j,i) = root_copy(j,i);
            dist_btwn_nodes(root(i,j),root(j,i))=...
               dbn_copy(root(i,j),root(j,i));
            dist_btwn_nodes(root(j,i),root(i,j))=...
               dbn_copy(root(i,j),root(j,i));
            tree_list= [tree_list;conn_nodes];
      else
            tree_list = conn_nodes;
      end
```

```
          con_nodes=not_conn;

    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% determine existence of all edges within clusters Inf means no edge
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

loc_total=0;
for k=1:numclusters
      if nodes_pc(k)~=1
            for i=loc_total+1:nodes_pc(k)-1+loc_total
                  for j=i+1:nodes_pc(k)+loc_total

                        if (rand > beta_within*exp(-dist_btwn_nodes...
                           (i,j)/(maxlength_within*alpha_within)))
                              dist_btwn_nodes(i,j)=0;
                        end

                        dist_btwn_nodes(j,i) = 0;
                  end
            end

%%%%%%%%%%%%%%%%%%%%%%%%%%
%Ensure no isolated nodes%
%%%%%%%%%%%%%%%%%%%%%%%%%%

            for i=loc_total+1:nodes_pc(k)+loc_total
                  if (nnz(dist_btwn_nodes(loc_total+1:i,i)==0))
                        temp=rand*maxlength_within;
                        if i ~=1+loc_total
                              dist_btwn_nodes(i-1,i)=temp;
                              dist_btwn_nodes(i,i-1) = temp;
                        else
                              dist_btwn_nodes(1+loc_total,2+loc_...
                                 total)=temp;
                              dist_btwn_nodes(2+loc_total,1+loc_...
                                 total)=temp;
                        end
                  end
            end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Ensure of no isolated fragments%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            not_conn=[loc_total+1:nodes_pc(k)+loc_total];
            tree_list=[];

            while length(tree_list) ~= nodes_pc(k)
                  conn_nodes=not_conn(1);
                  count=1;

                  while length(conn_nodes) >= count
```

```
                    temp=find(dist_btwn_nodes(conn_nodes(count),...
                          loc_total+1:nodes_pc(k)+loc_total)~=0) +...
                          loc_total;

                        for m=1:length(temp)
                            if any(conn_nodes==temp(m))
                                temp(m)=0;
                            end
                        end

                        if nonzeros(temp) ~= []
                            conn_nodes=[conn_nodes;nonzeros(temp)];
                        end

                        count=count+1;
                    end

                    if length(conn_nodes)~= length(not_conn)
                        not_conn=[];
                        for i=loc_total+1:nodes_pc(k)+loc_total
                            if any(conn_nodes == i)|any(tree_list==i)
                                ;
                            elseif not_conn==[]
                                not_conn=i;
                            else
                                not_conn=[not_conn;i];
                            end
                        end
                    end

                    if tree_list ~= []
                        i=tree_list(ceil(rand*length(tree_list)));
                        j=conn_nodes(ceil(rand*length(conn_nodes)));
                        dist_btwn_nodes(i,j) = 1;
                        dist_btwn_nodes(j,i) = dist_btwn_nodes(i,j);
                        tree_list= [tree_list;conn_nodes];
                    else
                        tree_list = conn_nodes;
                    end

                    con_nodes=not_conn;
                end

            end %if%

            loc_total=loc_total+nodes_pc(1,k);
        end

        if all(nodes_pc==1)
            avg_deg_within =0;
        else
            avg_deg_within=(nnz(dist_btwn_nodes)/(n-length...
              (find(nodes_pc==1))));
        end
```

```
fprintf('\nThe average node degree within a cluster is %f\n',...
   avg_deg_within)
avg_deg_btwn = nnz(root)/numclusters;
fprintf('The average node degree between clusters is %f\n',...
   avg_deg_btwn)


for i=1:n
      for j=1:n
            if dist_btwn_nodes(i,j) ~= 0

%stores the current reserved bandwidth as a uniform distribution
% between %0 and max bw for within a cluster.

                  bw_rsvd(i,j)=(rand*l_within);
                  bw_rsvd(j,i)=(rand*l_within);
            end
      end
end

%Overwrite the root entries (BR pairs)

for i=1:numclusters
      for j=1:numclusters
            if root(i,j) ~= 0
                  bw_rsvd(root(i,j), root(j,i)) = (rand*l_btwn);
                  bw_rsvd(root(j,i), root(i,j)) = (rand*l_btwn);
            end
      end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              FINDCENT.M                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function is required by thesisgraph.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% y = findcent(a,b) a and b are the nodes between which the
% intermediate node closest to the center is desired. The GLOBAL
% variables are the matrix of sp[n,n] which gives the shortest path
% length between any two nodes; the matrix hops gives the sequence of
% nodes between those two nodes; n is the total number of nodes; and d
% is the adjacency matrix for all nodes.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eric B. Boyer November 1993
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This file was modified from FINDCENT.M written by Boyer described
% above.
%
% Changes were introduced to accommodate asymmetric path costs in
% the cost_cor matrix. This first cut algorithm is the simplest approach
% that just examines the two paths and selects a center from the least
% cost.
%
% Participants are still considered both senders and receivers in this
% version. Therefore the adjacency matrix will store links to and from
% all intermediate hops.
%
% Modified by Eric Klinker, July 1994
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 2 August 3, 1994
% Eric Klinker
%
% This file was further modified by removing a previous modification
% that calculated the adjacency matrix for the tree.
%
% The previous version is stored as the file "findcent2.m" which is
% called from the script "findcore2.m".
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function y = findcent(a,b)

global sp cost_cor hops n


atob = sp(a,b);
btoa = sp(b,a);

if atob < btoa
      midcost = atob/2;
      dir = 1;
      hoprow = (a-1)*n+b;
      hopdst = a;
else
      midcost = btoa/2;
      dir = -1;
      hoprow = (b-1)*n+a;
      hopdst = b;
end

count = 1;
sumcost = 0;

while sumcost < midcost
      hopsrc = hopdst;
      if dir > 0 %a to b is shortest

            if hops(hoprow,count) == 0
                  hopdst = b;
            else
                  hopdst = hops(hoprow,count);
            end

            sumcost = sp(a, hopdst);
      else
            if hops(hoprow,count) == 0
                  hopdst = a;
            else
                  hopdst = hops(hoprow,count);
            end

            sumcost = sp(b, hopdst);
      end

      count = count + 1;
end


if sumcost - midcost == cost_cor(hopsrc,hopdst)/2
      y = min(hopsrc,hopdst);
elseif sumcost - midcost > cost_cor(hopsrc,hopdst)/2
      y = hopsrc;
else
      y = hopdst;
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                            FINDCORE.M                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is a script file for thesisgraph.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This script file will locate the core. It uses the Critical Set of
% Participants and finds the optimal core placement. It first pairs the
% CP's by putting the most distant CP's together. Each pair finds the
% intermediate node closest to the midpoint between them. This node is
% then paired with the midpoint of another pair. This process is
% continued until a single node is selected.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eric B. Boyer November 1993
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 2
% Modified by Eric Klinker 29 July 1994
%
% This version of findcore groups participants that may be only senders
% or only receivers, or both. It also calls FINDCENT which locates
% a center for a pair based on the traffic each of the pair handles
% and builds the appropriate adjacency matrix.
%
% The pairs and tourn data structure has changed to indicate the
% traffic direction associated with an entry. A number is stored
% after each entry. 1 is Send traffic only, 2 is Receive traffic only
% and 3 is both. A -1 indicates the traffic pattern for a bye. Entries
% in these vectors are now read in pairs.
%
% 30 July 1994
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% The following code pairs the participants based on the following
% rules:
%
% 1. First try to pair all senders to receivers starting the ones
% farthest apart.
% 2. If there are more receivers than senders, place all possible
% byes with receivers close to the cluster. Then pair all
% remaining receivers.
% 3. If there are more senders than receivers, place all possible
% byes with senders close to the cluster. Then pair all
% remaining senders.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 3 August 3, 1994
% Modified by Eric Klinker
%
% This code was modified to remove the adjacency matrix construction
% from findcent. Thus it calls findcent1 (the center location routine
% that does not build the adjacency matrix) and the data structures
% tourn and pairs were changed (again) to remove the direction entry
% after the node identifier.
%
% The old versions are now stored in the filenames "findcore2.m" and
% "findcent2.m" to preserve the tree that was build using that method.
% (although it was of little use).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pairs = []; totalcp = numsend + numrec;

senders_copy = senders;
receivers_copy = receivers;
odd_one_out = [];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The following code segment determines if there are more receivers than
% senders or vice versa and then pairs up particpants a minimum distance
% apart until the number of senders = the number of receivers remaining.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

diff = length(senders) - length(receivers);

if diff > 0
      sendsp = Zero2Inf(sp(senders,senders));

      while diff >= 2
            [m, ind1] = min(sendsp);
            [m, ind2] = min(m);
            pairs =[pairs,senders_copy(ind1(ind2)),senders_copy(ind2)];
            senders_copy([ind1(ind2) ind2]) = [];
            sendsp([ind1(ind2) ind2],:) =[];
            sendsp(:,[ind1(ind2) ind2]) =[];
            diff = diff -2;
      end

      if diff == 1
            [m, ind1] = min(sendsp);
            [m, ind2] = min(m);
            odd_one_out = [senders_copy(ind2)];
            senders_copy(ind2) = [];
      end

elseif diff < 0
      diff = abs(diff);
      recsp = Zero2Inf(sp(receivers, receivers));

      while diff >= 2
            [m, ind1] = min(recsp);
            [m, ind2] = min(m);
            pairs = [pairs, receivers_copy(ind1(ind2)), ...
               receivers_copy(ind2)];
            receivers_copy([ind1(ind2) ind2]) = [];
            recsp([ind1(ind2) ind2],:) =[];
            recsp(:,[ind1(ind2) ind2]) =[];
            diff = diff -2;
      end

      if diff == 1
            [m, ind1] = min(recsp);
            [m, ind2] = min(m);
            odd_one_out = [receivers_copy(ind2)];
            receivers_copy(ind2) = [];
      end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Now the # of Senders = # of receivers and the following code segment
% pairs those senders with the receivers that are the farthest away.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cpsp = sp(senders_copy, receivers_copy);

while ~isempty(senders_copy)
      [m, ind1] = max(cpsp);
      [m, ind2] = max(m);
      pairs = [pairs, senders_copy(ind1(ind2)),receivers_copy(ind2)];
      senders_copy(ind1(ind2)) = [];
      receivers_copy(ind2) = [];
      cpsp([ind1(ind2)],:) =[];

      if ~isempty(cpsp)
            cpsp(:,[ind2]) =[];
      end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This portion of code will properly locate the byes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% tourn will keep the addresses of the nodes involved
% and a value of -1 is the location of a bye
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% tourn was changed to hold the traffic type as well in this version
% k is the number of phases in the tournament. Byes are placed such
% that no participant gets subsequent byes in two phases of the tourn
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

k = ceil(log2(totalcp));
bye = 2^k - totalcp;
countb = 0;
tourn = zeros(k+1,2^k);

while bye > 0
      m = 2^floor(log2(bye));
      for i = 1:m
            tourn(1,2^(k-countb)-m+i) = -1;
      end
      bye = bye - m; countb = countb + 1;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This portion of code loads the pairs into the matrix
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% the vector "pairs" lists the addresses of the cp's as
% they are paired, the odd one out will be at the end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

countp = 0;

for i=1:2:2^k-1
      if tourn(1,i) ~= -1              % check if paired bye
            if tourn(1,i+1) ~= -1     % check if single bye
                  for j=1:2
                        tourn(1,i+j-1) = pairs(countp+j);
                  end
                  countp = countp + 2;
            else
                  tourn(1,i) = odd_one_out(1);
            end
      end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This portion of code will complete the selection
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Routine now calls findcent for asymmetric links (findcent.m)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


for phase = 1:k
      for i = 1:2:2^(k - phase + 1)  - 1

            if tourn(phase,i+1) == -1 | tourn(phase,i+1) ...
               == tourn(phase,i)
                  tourn(phase+1,(i+1)/2) = tourn(phase,i);
            else
                  tourn(phase+1,(i+1)/2) = ...
                     findcent(tourn(phase,i),tourn(phase,i+1));

            end
      end
end


core = tourn(k+1,1);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GET_COST.M
%
% This will take the bw_rsvd matrix and generate a new adjacency matrix,
% cost, which will be the actual cost of each link in the network.
%
% It will be derived from an equation where:
% Cost(i,j) = base_cost + exp (lambda*bw_rsvd(i,j)/max_bw)
% John Eichelberger April 1994
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Version 3 September 5 1994
% Eric Klinker
%
% Checked for any entry in bw_rsvd that was 0 and set it to INF before
% converting the matrix to a cost matrix. This eliminated 3 O(n^^2) loops
% following these calls in thesisgraph5.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cost=get_cost(bw_rsvd)
global lambda_within lambda_btwn nodes_pc basecost_within basecost_btwn
global root numclusters n l_within l_btwn

cost=zeros(n,n);

%modified by Eric Klinker to copy bw_rsvd matrix to cost matrix
% Left as a stub function for a later cost expression to be realized

for i = 1:n
      for j = 1:n

            if bw_rsvd(i,j) == 0     %added to replace the 3
                                     %loops in thesisgraph
                  bw_rsvd(i,j)=Inf;  %that did the same thing
            end

            cost(i,j)= bw_rsvd(i,j);
      end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                            GET_CPS.M                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This script file is part of thesisgraph.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This script file asks for the desired number of Critical Participants
% and then randomly locates them among the nodes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eric B. Boyer November 1993
% modified John Eichelberger April 1994
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Variables:
% cp_input - userlist of critical participants per cluster
% conc_send - total # of concurrent senders
% node_status - status vector to randomly determine critical
% participants
% csp - list of critical participants
% cp - count of critical participants
%
% Algorithm:
% - get # of critical participants for each cluster
% - get # of concurrent senders
% - randomly generate critical participants based on user input
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% File Modified by Eric Klinker 29 July 1994
%
% Modifications allow the user to specify the number of senders and
% receivers. The address of the senders are stored in the vector
% "senders" of size numsend. The receivers are stored in the vector
% "receivers" of size numrec.
%
% The senders and receivers are randomly dispersed throughout the
% clusters according to the vectors sender_input and receiver_input.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% gather sender input information

while 1
      fprintf ('\nThis is a list of nodes per cluster\n');
      disp(nodes_pc)
      fprintf('\nEnter # of Senders (vector form)\n');
      fprintf('Be sure to enter 1 entry for each node,even if zero.\n')
      fprintf('The # of Senders in each cluster will change...
         each iteration.\n')
      sender_input = input('-->');

      if all(sender_input >= 0)& length(sender_input)== ...
         length(nodes_pc) & all(sender_input==round(sender_input))
            break
      end
```

```
        fprintf('\n\nThe number of senders')
        fprintf('\nmust be a positive vector <= the number')
        fprintf('\nof nodes in the relative clusters.\n\n')
end

% gather receiver input information

while 1
        fprintf ('\nThis is a list of nodes per cluster\n');
        disp(nodes_pc)
        fprintf('\nEnter # of Receivers (vector form)\n');
        fprintf('Be sure to enter 1 entry for each node,even if zero.\n')
        fprintf('The # of Receivers in each cluster will change each...
           iteration.\n')
        receiver_input = input('-->');

        if all(receiver_input >= 0)& length(receiver_input)== ...
           length(nodes_pc) & all(receiver_input==round(receiver_input))
              break
        end

        fprintf('\n\nThe number of receivers')
        fprintf('\nmust be a positive vector <= the number')
        fprintf('\nof nodes in the relative clusters.\n\n')
end

while 1
        conc_send=input('Input # of concurrent senders -->');

        if conc_send <= sum(sender_input)
              break
        end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% locate the Senders randomly
% The vector senders[numsend] keeps the node address of each sender
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

loc_total=0;;
node_status=zeros(1,n);

for k=1:length(sender_input)
        clust=ceil(rand * numclusters);

        while loc_nodes_pc(clust)< sender_input(k)| clust > numclusters
              clust=ceil(rand * numclusters);
        end

        i=sender_input(k); loc_total=0;

        for j=1:k-1
              loc_total=loc_total+nodes_pc(j);
        end

        counter=1;

        while i ~=0
              if rand <= sender_input(k)/nodes_pc(k) & ...
                 node_status(loc_total+counter)==0
                    i=i-1;
                    node_status(loc_total+counter)=1;
                    loc_nodes_pc(clust) = loc_nodes_pc(clust)-1;
              end

              counter=counter+1;
              if counter>nodes_pc(k)
                    counter=1;
              end
        end

        loc_total=loc_total+nodes_pc(k);
end

senders=find(node_status~=0);
numsend = sum(sender_input);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% locate the Receivers randomly
% The vector receivers[numrec] keeps the node address of each CP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

loc_total=0;
loc_nodes_pc=nodes_pc;
node_status=zeros(1,n);

for k=1:length(receiver_input)
        clust=ceil(rand * numclusters);

        while loc_nodes_pc(clust)< receiver_input(k)| clust > numclusters
              clust=ceil(rand * numclusters);
        end

        i=receiver_input(k);
        loc_total=0;

        for j=1:k-1
              loc_total=loc_total+nodes_pc(j);
        end

        counter=1;

        while i ~=0
              if rand <= receiver_input(k)/nodes_pc(k) & ...
                 node_status(loc_total+counter)==0
                    i=i-1;
```

```
                        node_status(loc_total+counter)=1;
                        loc_nodes_pc(clust) = loc_nodes_pc(clust)-1;
                end
                counter=counter+1;
                if counter>nodes_pc(k)
                        counter=1;
                end
        end

        loc_total=loc_total+nodes_pc(k);
end

receivers=find(node_status~=0);
numrec = sum(receiver_input);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              GET_VALS.M                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is a script file that obtains the parameters required for
% thesisgraph.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eric B. Boyer November 1993
% modified John Eichelberger April 1994
% modified by Eric Klinker August 1994
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


defaults=input('defaults? (y or n)-->','s');

if defaults=='y'| defaults=='Y'
      alpha_btwn=.3;
      alpha_within=.3;
      beta_btwn=.6;
      beta_within=.8;
      n=10; l_btwn=5000;
      l_within=500;
      numclusters=4;
      rg_factor=2;
      gridarea_btwn=100;
      maxlength_btwn = sqrt(gridarea_btwn *2);
      maxlength_within = 2;
      basecost_btwn=1;
      basecost_within=1;
      lambda_btwn=2.93;
      lambda_within=2.93;
else

%%%%%%%%%%%%%
% get alpha %
%%%%%%%%%%%%%

      while 1
            fprintf('\nalpha is a relative value (0,1]')
            fprintf('\nwhere a small alpha will increase')
            fprintf('\nthe number of short edges relative')
            fprintf('\nto the number of long edges.\n')
            alpha_btwn = input('Enter the value for alpha for ...
               between clusters(.3): ');

            if alpha_btwn==[],
                  alpha_btwn=.3;
            end

            alpha_within = input('Enter the value for alpha for ...
               within clusters(.3): ');

            if alpha_within==[],
                  alpha_within=.3;
            end
```

```
        if alpha_btwn > 0 & alpha_within >0
              if alpha_btwn <=1 & alpha_within <=1
                    break
              end
        end

        fprintf('\n\nalpha needs to be > 0 and <= 1.\n')
  end


%%%%%%%%%%%%%
% get beta  %
%%%%%%%%%%%%%

  while 1
        fprintf('\n\nbeta is a relative value (0,1]')
        fprintf('\nthat is proportional to the number')
        fprintf('\nof edges.\n')
        beta_btwn = input('Enter the value for beta between...
           clusters(.6): ');
        beta_within = input('Enter the value for beta within ...
           clusters(.8): ');

        if beta_btwn==[],
              beta_btwn=.6;
        end

        if beta_within==[],
              beta_within=.8;
        end

        if beta_btwn > 0 & beta_within >0
              if beta_btwn <= 1 & beta_within <=1
                    break
              end
        end

        fprintf('\n\nbeta needs to be > 0 and <= 1.\n')
  end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get n and determine row and col dimensions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  while 1
        n = input('Enter the number of nodes(20): ');
        if n==[],
              n=20;
        end
        if n > 0
              if n == round(n)
                    break
              end
```

```
            end

            fprintf('\n\nThe number of nodes must be a positive...
               integer.\n')
        end

        xcor = ceil(4*sqrt(n/3));
        ycor = ceil(3*sqrt(n/3));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get maximum bandwidth
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        fprintf ('\n\nThe max bandwidth between nodes is defined as...
           300 within a cluster\n')
        fprintf ('and 3000 between clusters\n\n')
        l_btwn=5000;
        l_within=500;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get # of network clusters to simulate LAN's
% get avg # of nodes per cluster
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        while 1
              numclusters = input('Enter the number of network ...
                 clusters (min 2)(4): ');

              if numclusters==[],
                    numclusters=4;
              end

              if numclusters > 0
                    if numclusters == round(numclusters) |...
                       (n/numclusters ~= round(n/numclusters)) | ...
                       numclusters <2
                          break
                    end
              end

              fprintf('\n\nThe number of clusters must be a positive...
                 integer > 2\n')
        end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get model type for calculating distance between clusters
% (Within the clusters, the model will always be RG2 to simulate LAN)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        while 1
              fprintf ('\n\nDistance distribution within clusters:\n')
              fprintf ('1 - RG1\n2- RG2')
              rg_factor = input('3 - half of each (3)-->');
        if rg_factor==[],
            rg_factor=3;
        end

        if ((rg_factor > 0) & (rg_factor <= 3))
            maxlength_within = input ('Enter max distance btwn...
              nodes within clusters (2)-->');
            if maxlength_within ==[],
                maxlength_within = 2;
            end
            if rg_factor==2
                gridarea_btwn=[];
                maxlength_btwn =input('Enter the max ...
                  distance btwn clusters (10) -->');
                if maxlength_btwn==[],
                    maxlength_btwn=10;
                end
            else

                gridarea_btwn = input('Enter the square ...
                  area for RG1 placement (100) -->');
                if gridarea_btwn == [],
                    gridarea_btwn =100;
                end
                maxlength_btwn = sqrt(gridarea_btwn*2);
            end
            break
        end

        fprintf('\n\nThe number of nodes per cluster must be a...
          positive integer.\n')
    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get factors for calculating cost of links based on:
% cost = basecost + exp(lambda*bandwidth_reserved/bw_max)
% lambda will be calculated so that the overall cost will be 10X the
% base cost when bw_rsvd=75% of bw_max
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    fprintf ('\n\nInput base cost factor for equation:\n\n')
    fprintf (' link_cost = basecost + exp(lambda*bandwidth_reserved...
      /bw_max)\n')

    basecost_btwn=input('Base cost btwn clusters (10)-->');
    basecost_within=input('Base cost within clusters (1)-->');

    if basecost_btwn==[],
        basecost_btwn=10;
    end

    if basecost_within==[],
        basecost_within=1;
    end

    lambda_btwn= log(9*basecost_btwn+1)*(1/.75)
    lambda_within=log(9*basecost_within+1)+1*(1/.75)

end %if%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                             LAST_HOP.M                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function is for use in thesisgraph.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This function will find the last column of a vector that does not
% contain a padded zero.
% out = last_hop(in_hops) where in_hops is the input
% vector and out will be the column number.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eric B. Boyer November 1993
% modified by Eric Klinker August 1994
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function out = last_hop(in_hops);

out = length(in_hops);

while in_hops(out) == 0
      out = out - 1;
      if out == 0
            break
      end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              LINK_USE.M                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Function used by tree scripts to tally link usage by a particular
% session using a particular tree structure.
%
% The function utilizes the global list of senders and receivers.
% The function uses the hops and sp matrix for a given tree. The
% global variable hops should hold the intermediate hops for that
% particular tree. This requires that it be computed before calling
% LINK_USE.
%
% The function returns a n x n matrix link_cnt indicating the number
% of senders using a particular link.
%
% The function also returns a vector containing the average path
% length for each sender. This vector is avg_path_len.
%
% The function then calculates and returns an average path length
% (tree_path_len) for all senders of the tree.
%
% The program cycles through all senders to all receivers recording
% the links required for each sender and then totaling up the links
% used for all senders.
%
% Version 1 August 1, 1994
% Eric Klinker
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [link_cnt, avg_path_len, tree_path_len] = link_use

global senders receivers hops sp n

link_cnt = zeros(n,n);
avg_path_len = zeros(1,length(senders));
receiver_path_len = zeros(1,length(receivers));

for i = 1:length(senders)
      sender_cnt = zeros(n,n);
      for j = 1: length(receivers)

% for each sender, traverse its hop table
% counting the links used for each receiver

            hopsrc = senders(i);
            hopcnt = last_hop(hops((senders(i)-1)*n+receivers(j),:));

% Store the path length to each receiver in a vector

            receiver_path_len(j) = hopcnt+1;

            for k = 1:hopcnt+1
```

```
                if k == hopcnt + 1
                      hopdst = receivers(j);
                else
                      hopdst = hops((senders(i)-1)*n+receivers(j),k);
                end
                sender_cnt(hopsrc,hopdst)...
                   =sender_cnt(hopsrc,hopdst)+1; hopsrc = hopdst;
          end

    end %for j

    for l_row = 1:n
          for l_col = 1:n
                if sender_cnt(l_row,l_col) ~= 0
                      link_cnt(l_row,l_col) = link_cnt(l_row,l_col)+1;
                end
          end %for l_col
    end %for l_row


%Calculate the entry into avg_path_len for this sender.

avg_path_len(i) = sum(receiver_path_len)/length(receivers);

end %for i

tree_path_len = sum(avg_path_len)/length(senders);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              OUT_DATA.M                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This script file is for use in thesisgraph.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This file will output the data generated to graphs for analysis.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eric B. Boyer November 1993
% modified John Eichelberger April 1994
% modified by Eric Klinker August 1994
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


r=input('Which server printer do you want? [2,3,4,5,6 for file,...
   0 for no printed output] ');

if r~=0
      if r==6
            printer = ['print -dps -append ',input('filename:','s')];
      else
            printer = ['print -dps -Ppr1',int2str(r)];
      end
end

if r~=0
      eval(printer)
end

hold off

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure
clg
hold on
i=[1:iterations];
plot (i,cor_avg,'x')
plot (i,sh_avg,'o')
title('Average Path Length')
xlabel('# of interactions x = CBT, o = SBT, + = sparse, * = rp')
grid
if r~=0
      eval(printer)
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure
clg
hold on
plot (i,cor_conc_send,'x')
plot (i,sh_conc_send,'o')
```

```
title(['TREE COST for ' ,num2str(conc_send), ' concurrent senders'])
xlabel('# of interactions x = CBT, o = SBT, + = sparse,* = rp')
grid
if r~=0
      eval(printer)
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure
clg
hold on
i = [1:NumSend];
plot(i, crcst(i),'x')
plot(i,shcst(i),'o')
grid
title(['TREE COST vs # SIMULTANEOUS SENDERS on iteration',...
   num2str(iterations)])
xlabel('x = CBT, o = SBT, + = sparse, * = rp')
if r~=0
      eval(printer)
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure
clg
hold on
i = [1:NumSend];
plot(i, sendcst(i),'x')
plot(i,cor_sendcst(i),'o')
plot(i,reccst(i),'+')
plot(i,cor_reccst(i),'*')
grid
title(['Core Evaluation on iteration ',num2str(iterations)])
xlabel('x = SendCent, o = SendCore, + = RecCent, * = RecCore')
if r~=0
      eval(printer)
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              PLT_GRPH.M                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This script file is part of thesisgraph.m.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eric B. Boyer November 1993
% modified John Eichelberger April 1994
% modified by Eric Klinker August 1994
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Variables:
% x_dim, y_dim - dimension of plot grid (units of 1 cluster per
% square)
% dim - largest square necessary to plot all nodes of a cluster
% within
% spacing - counter for plotting nodes within a square
% graph_level - the horizontal row of the plotting grid
% graph_column - the vertical column of the plotting grid
% x_ticks,y_ticks - used to properly space grid marks for MATLAB
% plotting
% Algorithm:
% - generate a grid plot based on numclusters (x_dim,y_dim)
% - generate matrix to send to gplot
% - space nodes within square associated with particular
% cluster
% - plot 'o' at node location on grid
% - plot gridlines to visually divide clusters
% - plot links btwn clusters in blue
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

xy=zeros(n,2);

temp=numclusters;
x_dim=1;

while temp/3==round(temp/3) & temp > x_dim
      x_dim=x_dim*3;
      temp=temp/3;
end

while temp/2==round(temp/2) & temp >x_dim
      x_dim=x_dim*2;
      temp=temp/2;
end

temp2= ceil(sqrt(numclusters));
if (x_dim <= temp2 & temp <= temp2) | (temp2*2==x_dim+temp)
      y_dim=temp;
else
      x_dim=temp2;
      y_dim=temp2;
end
```

```
loc_total=0;
temp=1;
temp2=1;
graph_level=1;
graph_column=0;
dim=ceil(sqrt(max(nodes_pc)));

for k=1:numclusters
      spacing= ((dim-1)^2/nodes_pc(k))+(1/nodes_pc(k));
      for i=1:nodes_pc(k)
            while temp2>graph_level*dim
                  temp=temp+.7;
                  temp2=temp2-dim;
            end
            if nodes_pc(k) ==1
                  xy(loc_total+i,:)=[(temp+dim/2-1) (temp2+dim/2-1)];
            else
                  xy(loc_total+i,:)=[temp temp2];
            end
            temp2=temp2+spacing;
      end

      loc_total=loc_total+nodes_pc(k);
      if graph_column==x_dim-1
            graph_column=0;
            temp2=dim*graph_level+1;
            graph_level=graph_level+1;
      else
            temp2=dim*(graph_level-1)+1;
            graph_column=graph_column + 1;
      end

      temp=dim*graph_column +1;
end

x_ticks=([1:x_dim]/(x_dim));
y_ticks=([1:y_dim]/(y_dim));
clg
hold on
gplot(dist_btwn_nodes,xy,'r:')
text (xy(:,1),xy(:,2),'o')

%plot between cluster links in blue%

for i=1:numclusters-1
      for j=i:numclusters
            if root(i,j) ~=0
                  line ([xy(root(i,j),1) xy(root(j,i),1)],...
                     [xy(root(i,j),2) xy(root(j,i),2)],'Color',[0 0 1])
            end
      end
end

axis([0 (x_dim*dim) 0 (y_dim*dim)])
```

```
text(0,y_dim*dim,['WITHIN CLUSTERS: alpha = ', num2str(alpha_within)
, ... 'beta =',num2str(beta_within), ' max cost = ', int2str(l_within)])
axis off
axes('XTick',x_ticks,'YTick',y_ticks)
grid
set (gca,'XTickLabels',[],'YTickLabels',[])

title(['BETWEEN CLUSTERS: alpha = ', num2str(alpha_btwn), ' beta = ',...
   num2str(beta_btwn), ' max cost = ', int2str(l_btwn), ' nodes = '
,... int2str(n)])

xlabel(['avg node deg within clusters= ',num2str(avg_deg_within)])
ylabel(['avg node deg btwn clusters= ',num2str(avg_deg_btwn)])
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              SHRTPATH.M                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function is for use in thesisgraph.m.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% [sp, hops] = shrtpath(a, d)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% shrtpath input will be source node address (a) and the adjacency
% matrix d[n,n]. Output will be sp[n] which is a vector of distances
% from the source to all other nodes and hops[n,n-2] which is the
% intermediate nodes between the source and the destination. Note that
% hops is padded with zeros to make all row vectors have a length
% of n-2.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Eric B. Boyer November 1993
% modified by Eric Klinker August 1994
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [sp, hops] = shrtpath(a, cost)
n = size(cost,1);

sp=Inf*ones(1,n);
parent = a*ones(1,n); hops = [];

sp(a) = 0;
ntally = 1:n;

while length(ntally) > 1
      [m,i] = min(sp(ntally));
      checkwho = ntally(i);
      ntally(i) = [];
      dsp = cost(checkwho,ntally);

      for i = 1:length(ntally)
            if m + dsp(i) < sp(ntally(i))
                  p(ntally(i)) = m + dsp(i) ;
                  parent(ntally(i)) = checkwho;
            end
      end

end

for j = 1:n
      hopping = parent(j);

      while hopping(1) ~= a
            hopping = [parent(hopping(1)), hopping];
      end

      hopping(1) = [];
```

```
    hopping = [hopping, zeros(1, n-2 - length(hopping))];
    hops = [hops; hopping];
end
```

LIST OF REFERENCES

[1] Tony Ballardie, Paul Francis, and Jon Crowcroft, "Core based trees (CBT) an architecture for scalable inter-domain multicast routing," in *ACM SIGCOMM*, September 1993.

[2] Stephen Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei, *Protocol Independent Multicast (PIM): Motivation and Architecture*, Internet Draft, draft-ietf-idmr-pim-arch-00.ps, March 1994.

[3] Stephen Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei, *Protocol Independent Multicast (PIM), Sparse Mode Protocol Specification*, Internet Draft, draft-ietf-idmr-pim-sparse-spec-00.ps, March 1994.

[4] Stephen Deering, Deborah Estrin, Dino Farinacci, and Van Jacobson, *Protocol Independent Multicast (PIM), Dense Mode Protocol Specification,* Internet Draft, draft-ietf-idmr-pim-dense-spec-00.txt, March 1994.

[5] D. Waitzman, C. Partridge, and S. Deering, *Distance Vector Multicast Routing Protocol*, Technical Report RFC 1075, Internet, Network Working Group, November 1988.

[6] J. Moy, *OSPF Version 2*, Technical Report RFC 1247, Internet, Network Working Group, July 1991.

[7] J. Moy, *Multicast Extensions to OSPF*, Technical Report RFC 1584, Network Working Group, March 1994.

[8] Stephen Deering and David Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM Transactions on Computer Systems*, May 1990.

[9] Bernard M. Waxman, "Routing of Multipoint Connections," *IEEE Selected Areas in Communications*, December 1988.

[10] David D. Clark, Scott Shenker, and Lixia Zhang, "Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism," *ACM SIGCOMM*, September 1992.

[11] Stephen Deering, *Multicast Routing in a Datagram Network*, Ph.D. thesis, December 1991.

[12] Liming Wei and Deborah Estrin, *A Comparison of Multicast Trees and Algorithms*, Draft Submitted to INFOCOM 1994, 1993.

[13] Stephen E. Deering, *Host Extensions for IP Multicasting*, Technical Report RFC 1112, Internet, Network Working Group, August 1989.

[14] L. Zhang, R Braden, D Estrin, S Herzog, and S. Jamin, *Resource ReSerVation Protocol (RSVP)- Version 1 Functional Specification*, Internet Draft, draft-ietf-rsvp-spec-03.ps, July 1994.

[15] Robert J. Barton, *Multicast Analysis Simulation Tool (MAST)*, Technical Report, Naval Postgraduate School, September 1994.

[16] Van Jacobson, *The Session Directory Tool*, Lawrence Berkeley Laboratory, 1992.

[17] Y. Rekhter and T. Li, editors. *A Border Gateway Protocol 4 (BGP-4)*, Internet Draft, January 1994.

[18] S. Hares and John Scudder, *IDRP for IP*, Internet Draft, September 1993.

[19] D. Estrin, T. Li, Y. Rekhter, and D. Zappala, *Source demand Routing Protocol: Packet Format and Forwarding Specification*, Internet Draft, March 1993.

[20] K. Claffy, G. Polyzos, and H. W. Braun, "Traffic Characteristics of the T1 NSFNET Backbone," *Proceedings of INFOCOM '93*, March 1993.

[21] Merit Inc., Statistics available via anonymous ftp to nic.merit.edu, directory nsfnet/statistics; May 1994.

[22] Vern Paxson, "Growth Trends in Wide-Area TCP Connections," *IEEE Network*, August 1994.

[23] Eric Boyer, *Multicast Communication With Guaranteed Quality of Service*, Master's Thesis, Naval Postgraduate School, December 1993.

BIBLIOGRAPHY

[1] John Eichelberger, *Expanding on Multicast Communication With Guaranteed Quality of Service,* work in progress, Naval Postgraduate School, April 1994.

[2] Shridhar Shukla and Eric Boyer, "A Scalable Approach for Rapid Construction of Multicast Trees with Guaranteed Quality of Service", submitted to the *IEEE/ACM Transactions on Networking,* January 1994.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5101	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
4.	Professor Shridhar B. Shukla Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	2
5.	Professor Gilbert Lundy Code CS/Ln Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5118	1
6.	James Eric Klinker Code 5544 Naval Research Laboratory 4555 Overlook Ave. SW Washington, D.C.20375-5000	2